

new/exception_lists/packaging

1

```
*****
26867 Sat Aug 18 10:36:54 2012
new/exception_lists/packaging
dccc: finish move headers, cleanup dccc states
*****

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 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25 #

27 #
28 # Exception List for validate_pkg
29 #

31 #
32 # The following entries are built in the /proto area
33 # but not included in any packages - this is intentional.
34 #
35 usr/include/auth_list.h
36 usr/include/bsm/audit_door_infc.h
37 usr/include/bsm/audit_private.h
38 usr/include/bsm/devalloc.h
39 usr/include/getxby_door.h
40 usr/include/passwdutil.h
41 usr/include/priv_utils.h
42 usr/include/rpcsvc/daemon_utils.h
43 usr/include/rpcsvc/svc_dg_priv.h
44 usr/include/security/pam_impl.h
45 usr/include/sys/clock_impl.h
46 usr/include/sys/ieeeep.h
47 usr/include/sys/winlockio.h
48 usr/include/scsi/plugins/ses/vendor/sun_impl.h
49 #
50 # Private/Internal libraries of the Cryptographic Framework.
51 #
52 lib/libkcf.so
53 lib/lib-lelfsign
54 lib/lib-lelfsign.ln
55 lib/lib-lkcf
56 lib/lib-lkcf.ln
57 usr/include/libelfsign.h
58 usr/lib/lib-lsoftcrypto
59 usr/lib/lib-lsoftcrypto.ln
60 usr/lib/amd64/lib-lsoftcrypto.ln      i386
61 usr/lib/sparcv9/lib-lsoftcrypto.ln    sparv
```

new/exception_lists/packaging

2

```
63 #
64 # The following files are used by the DHCP service, the
65 # standalone's DHCP implementation, and the kernel (nfs_dlboot).
66 # They contain interfaces which are currently private.
67 #
68 usr/include/dhcp_svc_confkey.h
69 usr/include/dhcp_svc_confopt.h
70 usr/include/dhcp_svc_private.h
71 usr/include/dhcp_symbol.h
72 usr/include/sys/sunos_dhcp_class.h
73 usr/lib/libdhcpsvc.so
74 usr/lib/lib-ldhcpsvc
75 usr/lib/lib-ldhcpsvc.ln
76 #
77 # Private MAC driver header files
78 #
79 usr/include/inet/iptun.h
80 usr/include/sys/aggr_impl.h
81 usr/include/sys/aggr.h
82 usr/include/sys/dld_impl.h
83 usr/include/sys/dld_ioc.h
84 usr/include/sys/dls_impl.h
85 usr/include/sys/dls.h
86 usr/include/sys/mac_client_impl.h
87 usr/include/sys/mac_client.h
88 usr/include/sys/mac_flow_impl.h
89 usr/include/sys/mac_impl.h
90 usr/include/sys/mac_soft_ring.h
91 usr/include/sys/mac_stat.h
92 #
93 # Private GLDv3 userland libraries and headers
94 #
95 usr/include/libdladm_impl.h
96 usr/include/libdlaggr.h
97 usr/include/libdlether.h
98 usr/include/libdlflow_impl.h
99 usr/include/libdlflow.h
100 usr/include/libdliptun.h
101 usr/include/libdlmgmt.h
102 usr/include/libdlsim.h
103 usr/include/libdlstat.h
104 usr/include/libdlvnic.h
105 usr/include/libdlwlan_impl.h
106 usr/include/libdlwlan.h
107 #
108 # Virtual Network Interface Card (VNIC)
109 #
110 usr/include/sys/vnic.h
111 usr/include/sys/vnic_impl.h
112 #
113 # Private libipadm lint library and header files
114 #
115 usr/include/ipadm_ipmgmt.h
116 usr/include/ipadm_ndpd.h
117 usr/include/libipadm.h
118 lib/lib-lipadm
119 lib/lib-lipadm.ln
120 lib/libipadm.so
121 #
122 # Private libsocket header file
123 #
124 usr/include/libsocket_priv.h
125 #
126 # IKE and IPsec support library exceptions. The IKE support
127 # library contains exclusively private interfaces, as does
```

new/exception_lists/packaging

3

```

128 # libipsecutil. My apologies for the glut of header files here.
129 #
130 usr/include/errfp.h
131 usr/include/ikedoor.h
132 usr/include/ipsec_util.h
133 usr/lib/libike.so
134 usr/lib/amd64/libike.so          i386
135 usr/lib/sparcv9/libike.so       sparc
136 usr/lib/libipsecutil.so
137 usr/lib/amd64/libipsecutil.so   i386
138 usr/lib/sparcv9/libipsecutil.so sparc
139 usr/lib/liblike.so
140 usr/lib/liblike.ln
141 usr/lib/amd64/liblike.ln       i386
142 usr/lib/sparcv9/liblike.ln     sparc
143 usr/lib/liblipsecutil.so
144 usr/lib/liblipsecutil.ln
145 usr/lib/amd64/liblipsecutil.ln i386
146 usr/lib/sparcv9/liblipsecutil.ln sparc
147 #
148 usr/include/inet/dccp_impl.h
149 #endif /* ! codereview */
150 usr/include/inet/ip_impl.h
151 usr/include/inet/ip_ndp.h
152 usr/include/inet/ip2mac_impl.h
153 usr/include/inet/ip2mac.h
154 usr/include/inet/rawip_impl.h
155 usr/include/inet/tcp_impl.h
156 usr/include/inet/udp_impl.h
157 usr/include/libmail.h
158 usr/include/libnwam_priv.h
159 usr/include/protocols/ripngd.h
160 usr/include/s_string.h
161 usr/include/sys/logindmux_impl.h
162 usr/include/sys/vgareg.h
163 #
164 # Some IPsec headers can't be shipped lest we hit export controls...
165 #
166 usr/include/inet/ipsec_impl.h
167 usr/include/inet/ipsec_info.h
168 usr/include/inet/ipsec.h
169 usr/include/inet/ipsecesp.h
170 usr/include/inet/keysock.h
171 usr/include/inet/sadb.h
172 usr/include/sys/sha1_consts.h
173 usr/include/sys/sha2_consts.h
174 #
175 #
176 # Filtering out directories not shipped
177 #
178 usr/4lib          i386
179 #
180 # These files contain definitions shared privately between the kernel
181 # and libc. There is no reason for them to be part of a package that
182 # a customer should ever see. They are installed in the proto area by
183 # the uts build because libc and other components, like truss, are
184 # dependent upon their contents and should not have their own copies.
185 #
186 usr/include/sys/libc_kernel.h
187 usr/include/sys/synch32.h
188 #
189 # These files are installed in the proto area by the build of libproc for
190 # the benefit of the builds of cmd/truss, cmd/gcore and cmd/ptools, which
191 # use libproc as their common process-control library. These are not
192 # interfaces for customer use, so the files are excluded from packaging.
193 #

```

new/exception_lists/packaging

4

```

194 lib/liblproc
195 lib/liblproc.ln
196 lib/amd64/liblproc.ln          i386
197 lib/sparcv9/liblproc.ln       sparc
198 usr/include/libproc.h
199 #
200 # Private interfaces for libdisasm
201 #
202 usr/include/libdisasm.h
203 usr/lib/libl-disasm
204 usr/lib/libl-disasm.ln
205 usr/lib/amd64/libl-disasm.ln   i386
206 usr/lib/sparcv9/libl-disasm.ln sparc
207 #
208 # Private interfaces for libraidcfg
209 #
210 usr/include/raidcfg_spi.h
211 usr/include/raidcfg.h
212 usr/lib/libraidcfg.so
213 usr/lib/amd64/libraidcfg.so    i386
214 usr/lib/sparcv9/libraidcfg.so  sparc
215 usr/lib/libl-raidcfg
216 usr/lib/libl-raidcfg.ln
217 usr/lib/amd64/libl-raidcfg.ln  i386
218 usr/lib/sparcv9/libl-raidcfg.ln sparc
219 #
220 # This file is used for private communication between mdb, drv/kmdb, and
221 # misc/kmdb. The interfaces described herein are not intended for customer
222 # use, and are thus excluded from packaging.
223 #
224 usr/include/sys/kmdb.h
225 #
226 # These files are installed in the proto area by the build of libdhcpgent
227 # and libdhcputil for the benefit of DHCP-related networking commands such
228 # as dhcpgent, dhcpcinfo, ifconfig, and netstat. These are not interfaces
229 # for customer use, so the files are excluded from packaging.
230 #
231 lib/libdhcpgent.so
232 lib/libdhcputil.so
233 lib/libl-dhcpgent
234 lib/libl-dhcpgent.ln
235 lib/libl-dhcputil
236 lib/libl-dhcputil.ln
237 usr/include/dhcp_hostconf.h
238 usr/include/dhcp_impl.h
239 usr/include/dhcp_inittab.h
240 usr/include/dhcp_stable.h
241 usr/include/dhcp_symbol_common.h
242 usr/include/dhcpgent_ipc.h
243 usr/include/dhcpgent_util.h
244 usr/include/dhcpmsg.h
245 usr/lib/libdhcpgent.so
246 usr/lib/libdhcputil.so
247 usr/lib/libl-dhcpgent
248 usr/lib/libl-dhcpgent.ln
249 usr/lib/libl-dhcputil
250 usr/lib/libl-dhcputil.ln
251 #
252 # These files are installed in the proto area by the build of libinstzones
253 # and libpkg
254 #
255 usr/lib/libl-linstzones
256 usr/lib/libl-linstzones.ln
257 usr/lib/libl-lpkg
258 usr/lib/libl-lpkg.ln
259 #

```

```

260 # Don't ship header files private to libipmp and in.mpathd
261 #
262 usr/include/ipmp_query_impl.h
263 #
264 # These files are installed in the proto area by the build of libinetsvc,
265 # an inetd-specific library shared by inetd, inetadm and inetconv. Only
266 # the shared object is shipped.
267 #
268 usr/include/inetsvc.h
269 usr/lib/libinetsvc.so
270 usr/lib/llib-libinetsvc
271 usr/lib/llib-libinetsvc.ln
272 #
273 # These files are installed in the proto area by the build of libinetutil,
274 # a general purpose library for the benefit of internet utilities. Only
275 # the shared object is shipped.
276 #
277 lib/libinetutil.so
278 lib/amd64/libinetutil.so          i386
279 lib/sparcv9/libinetutil.so       sparc
280 lib/llib-libinetutil
281 lib/llib-libinetutil.ln
282 lib/amd64/llib-libinetutil.ln   i386
283 lib/sparcv9/llib-libinetutil.ln sparc
284 usr/include/libinetutil.h
285 usr/include/netinet/inetutil.h
286 usr/include/ofmt.h
287 usr/lib/libinetutil.so
288 usr/lib/amd64/libinetutil.so     i386
289 usr/lib/sparcv9/libinetutil.so   sparc
290 usr/lib/llib-libinetutil
291 usr/lib/llib-libinetutil.ln
292 usr/lib/amd64/llib-libinetutil.ln i386
293 usr/lib/sparcv9/llib-libinetutil.ln sparc
294 #
295 # Miscellaneous kernel interfaces or kernel<->user interfaces that are
296 # consolidation private and we do not want to export at this time.
297 #
298 usr/include/sys/cryptmod.h
299 usr/include/sys/dumpadm.h
300 usr/include/sys/ontrap.h
301 usr/include/sys/sysmsg_impl.h
302 usr/include/sys/vlan.h
303 #
304 # These files are installed in the proto area so lvm can use
305 # them during the build process.
306 #
307 lib/llib-lmeta
308 lib/llib-lmeta.ln
309 usr/include/sdssc.h
310 usr/lib/llib-lmeta
311 usr/lib/llib-lmeta.ln
312 #
313 # non-public pci header
314 #
315 usr/include/sys/pci_impl.h
316 usr/include/sys/pci_tools.h
317 #
318 # Exception list for RCM project, included by librcm and rcm_daemon
319 #
320 usr/include/librcm_event.h
321 usr/include/librcm_impl.h
322 #
323 # MDB deliverables that are not yet public
324 #
325 usr/lib/mdb/proc/mdb_test.so

```

```

326 usr/lib/mdb/proc/sparcv9/mdb_test.so    sparc
327 #
328 # SNCA project exception list
329 #
330 usr/include/inet/kssl/kssl.h
331 usr/include/inet/kssl/ksslimpl.h
332 usr/include/inet/kssl/ksslproto.h
333 usr/include/inet/nca
334 #
335 # these are "removed" from the source product build because the only
336 # packages that currently deliver them are removed.
337 # they really should't be in here.
338 #
339 etc/sfw
340 #
341 # Entries for the libmech_krb5 symlink, which has been included
342 # for build purposes only, not delivered to customers.
343 #
344 usr/lib/gss/libmech_krb5.so
345 usr/lib/amd64/gss/libmech_krb5.so        i386
346 usr/lib/sparcv9/gss/libmech_krb5.so     sparc
347 usr/lib/libmech_krb5.so
348 usr/lib/amd64/libmech_krb5.so           i386
349 usr/lib/sparcv9/libmech_krb5.so         sparc
350 #
351 # Entries for headers from efcode project which user does not need to see
352 #
353 usr/platform/sun4u/include/sys/fc_plat.h      sparc
354 usr/platform/sun4u/include/sys/fcode.h        sparc
355 #
356 # Private net80211 headers
357 #
358 usr/include/sys/net80211_crypto.h
359 usr/include/sys/net80211_ht.h
360 usr/include/sys/net80211_proto.h
361 usr/include/sys/net80211.h
362 #
363 usr/include/net/wpa.h
364 #
365 # PPPoE files not delivered to customers.
366 #
367 usr/include/net/pppoe.h
368 usr/include/net/sppptun.h
369 #
370 # Simnet
371 #
372 usr/include/net/simnet.h
373 #
374 # Bridging internal data structures
375 #
376 usr/include/net/bridge_impl.h
377 #
378 # User<->kernel interface used by cfgadm/USB only
379 #
380 usr/include/sys/usb/hubd/hubd_impl.h
381 #
382 # User<->kernel interface used by cfgadm/SATA only
383 #
384 usr/include/sys/sata/sata_cfgadm.h          i386
385 #
386 # Private ucred kernel header
387 #
388 usr/include/sys/ucred.h
389 #
390 # Private and/or platform-specific smf(5) files
391 #

```

```

392 lib/librestart.so
393 lib/lib-lirestart
394 lib/lib-lirestart.ln
395 lib/amd64/lib-lirestart.ln      i386
396 lib/sparcv9/lib-lirestart.ln    sparc
397 usr/include/libcontract_priv.h
398 usr/include/librestart_priv.h
399 usr/include/librestart.h
400 usr/lib/librestart.so
401 usr/lib/sparcv9/librestart.so     sparc
402 lib/svc/manifest/platform/sun4u  i386
403 lib/svc/manifest/platform/sun4v  i386
404 var/svc/manifest/platform/sun4u  i386
405 var/svc/manifest/platform/sun4v  i386
406 etc/svc/profile/platform_sun4v.xml i386
407 etc/svc/profile/platform_SUNW,SPARC-Enterprise.xml i386
408 etc/svc/profile/platform_SUNW,Sun-Fire-15000.xml i386
409 etc/svc/profile/platform_SUNW,Sun-Fire-880.xml i386
410 etc/svc/profile/platform_SUNW,Sun-Fire-V890.xml i386
411 etc/svc/profile/platform_SUNW,Sun-Fire.xml i386
412 etc/svc/profile/platform_SUNW,Ultra-Enterprise-10000.xml i386
413 etc/svc/profile/platform_SUNW,UltraSPARC-IIe-NetraCT-40.xml i386
414 etc/svc/profile/platform_SUNW,UltraSPARC-IIe-NetraCT-60.xml i386
415 etc/svc/profile/platform_SUNW,UltraSPARC-IIi-Netra.xml i386
416 #
417 # Private libuutil files
418 #
419 lib/libuutil.so
420 lib/lib-luutil
421 lib/lib-luutil.ln
422 lib/sparcv9/lib-luutil.ln      sparc
423 usr/include/libuutil_impl.h
424 usr/lib/libuutil.so
425 usr/lib/sparcv9/libuutil.so     sparc
426 #
427 # Private Multidata file.
428 #
429 usr/include/sys/multidata_impl.h
430 #
431 # The following files are used by wanboot.
432 # They contain interfaces which are currently private.
433 #
434 usr/include/sys/wanboot_impl.h
435 usr/include/wanboot
436 usr/include/wanbootutil.h
437 #
438 # Even though all the objects built under usr/src/stand are later glommed
439 # together into a couple of second-stage boot loaders, we dump the static
440 # archives and lint libraries into $(ROOT)/stand for intermediate use
441 # (e.g., for lint, linking the second-stage boot loaders, ...). Since
442 # these are merely intermediate objects, they do not need to be packaged.
443 #
444 stand                          sparc
445 #
446 # Private KCF header files
447 #
448 usr/include/sys/crypto/elfsign.h
449 usr/include/sys/crypto/impl.h
450 usr/include/sys/crypto/ops_impl.h
451 usr/include/sys/crypto/sched_impl.h
452 #
453 # The following files are installed in the proto area
454 # by the build of libavl (AVL Tree Interface Library).
455 # libavl contains interfaces which are all private interfaces.
456 #
457 lib/libavl.so

```

```

458 lib/amd64/libavl.so            i386
459 lib/sparcv9/libavl.so          sparc
460 lib/lib-lavl
461 lib/lib-lavl.ln
462 lib/amd64/lib-lavl.ln          i386
463 lib/sparcv9/lib-lavl.ln        sparc
464 usr/lib/libavl.so
465 usr/lib/amd64/libavl.so        i386
466 usr/lib/sparcv9/libavl.so      sparc
467 usr/lib/lib-lavl
468 usr/lib/lib-lavl.ln
469 usr/lib/amd64/lib-lavl.ln      i386
470 usr/lib/sparcv9/lib-lavl.ln    sparc
471 #
472 # The following files are installed in the proto area
473 # by the build of libcmdutils (Command Utilities Library).
474 # libcmdutils contains interfaces which are all private interfaces.
475 #
476 lib/libcmdutils.so
477 lib/amd64/libcmdutils.so        i386
478 lib/sparcv9/libcmdutils.so      sparc
479 lib/lib-lcmdutils
480 lib/lib-lcmdutils.ln
481 lib/amd64/lib-lcmdutils.ln      i386
482 lib/sparcv9/lib-lcmdutils.ln    sparc
483 usr/include/libcmdutils.h
484 usr/lib/libcmdutils.so
485 usr/lib/amd64/libcmdutils.so     i386
486 usr/lib/sparcv9/libcmdutils.so   sparc
487 usr/lib/lib-lcmdutils
488 usr/lib/lib-lcmdutils.ln
489 usr/lib/amd64/lib-lcmdutils.ln   i386
490 usr/lib/sparcv9/lib-lcmdutils.ln sparc
491 #
492 # Private interfaces in libsec
493 #
494 usr/include/aclutils.h
495 #
496 # USB skeleton driver stays in sync with the rest of USB but doesn't ship.
497 #
498 kernel/drv/usbkskel            i386
499 kernel/drv/amd64/usbkskel       i386
500 kernel/drv/sparcv9/usbkskel     sparc
501 kernel/drv/usbkskel.conf
502 #
503 # Consolidation and Sun private libdevid interfaces
504 # Public libdevid interfaces provided by devid.h
505 #
506 usr/include/sys/libdevid.h
507 #
508 # The following files are installed in the proto area by the build of
509 # libprtdiag. libprtdiag contains interfaces which are all private.
510 # Only the shared object is shipped.
511 #
512 usr/platform/sun4u/lib/lib-lprtdiag      sparc
513 usr/platform/sun4u/lib/lib-lprtdiag.ln   sparc
514 usr/platform/sun4v/lib/lib-lprtdiag.ln   sparc
515 #
516 # The following files are installed in the proto area by the build of
517 # mdesc driver in sun4v. These header files are used on in the build
518 # and do not need to be shipped to customers.
519 #
520 usr/include/sys/mdesc.h                  sparc
521 usr/include/sys/mdesc_impl.h             sparc
522 usr/platform/sun4v/include/sys/mach_descrip.h sparc
523 #

```

```

524 # The following files are installed in the proto area by the build of
525 # libpcp. libpcp contains interfaces which are all private.
526 # Only the shared object is shipped.
527 #
528 usr/platform/sun4v/lib/lib-lpcp.ln          sparc
529 usr/platform/SUNW,Netra-CP3060/lib/lib-lpcp.ln sparc
530 usr/platform/SUNW,Netra-CP3260/lib/lib-lpcp.ln sparc
531 usr/platform/SUNW,Netra-T5220/lib/lib-lpcp.ln sparc
532 usr/platform/SUNW,Netra-T5440/lib/lib-lpcp.ln sparc
533 usr/platform/SUNW,SPARC-Enterprise-T5120/lib/lib-lpcp.ln sparc
534 usr/platform/SUNW,Sun-Blade-T6300/lib/lib-lpcp.ln sparc
535 usr/platform/SUNW,Sun-Blade-T6320/lib/lib-lpcp.ln sparc
536 usr/platform/SUNW,Sun-Fire-T200/lib/lib-lpcp.ln sparc
537 usr/platform/SUNW,T5140/lib/lib-lpcp.ln      sparc
538 usr/platform/SUNW,USBRDT-5240/lib/lib-lpcp.ln sparc
539 #
540 # ZFS internal tools and lint libraries
541 #
542 usr/lib/lib-lzfs_jni
543 usr/lib/lib-lzfs_jni.ln
544 usr/lib/amd64/lib-lzfs_jni.ln          i386
545 usr/lib/sparcv9/lib-lzfs_jni.ln       sparc
546 usr/lib/lib-lzpool
547 usr/lib/lib-lzpool.ln                  i386
548 usr/lib/amd64/lib-lzpool.ln           i386
549 usr/lib/sparcv9/lib-lzpool.ln         sparc
550 #
551 # ZFS JNI headers
552 #
553 usr/include/libzfs_jni_dataset.h
554 usr/include/libzfs_jni_disk.h
555 usr/include/libzfs_jni_diskmt.h
556 usr/include/libzfs_jni_ipool.h
557 usr/include/libzfs_jni_main.h
558 usr/include/libzfs_jni_pool.h
559 usr/include/libzfs_jni_property.h
560 usr/include/libzfs_jni_util.h
561 #
562 # These files are installed in the proto area for Solaris scsi_vhci driver
563 # (for MPAPI support) and should not be shipped
564 #
565 usr/include/sys/scsi/adapters/mpapi_impl.h
566 usr/include/sys/scsi/adapters/mpapi_scsi_vhci.h
567 #
568 # This library is installed in the proto area by the build of libdisasm, and is
569 # only used when building the KMDB disasm module.
570 #
571 usr/lib/libstanddisasm.so
572 usr/lib/amd64/libstanddisasm.so        i386
573 usr/lib/sparcv9/libstanddisasm.so      sparc
574 #
575 # TSol: tsol doesn't ship lint source, and tsnet isn't for customers at all.
576 #
577 lib/libtsnet.so
578 usr/lib/lib-ltsnet
579 usr/lib/lib-ltsol
580 #
581 # nss interfaces shared between libnsl and other ON libraries.
582 #
583 usr/include/nss.h
584 #
585 # AT&T AST (ksh93) files which are currently needed only to build OS/Net
586 # (msgcc&co.)
587 # libast
588 usr/lib/libast.so
589 usr/lib/amd64/libast.so                i386

```

```

590 usr/lib/sparcv9/libast.so              sparc
591 usr/lib/lib-last
592 usr/lib/lib-last.ln
593 usr/lib/amd64/lib-last.ln             i386
594 usr/lib/sparcv9/lib-last.ln          sparc
595 # libcmd
596 usr/lib/lib-lcmd
597 usr/lib/lib-lcmd.ln
598 usr/lib/amd64/lib-lcmd.ln             i386
599 usr/lib/sparcv9/lib-lcmd.ln          sparc
600 # libdll
601 usr/lib/libdll.so
602 usr/lib/amd64/libdll.so               i386
603 usr/lib/sparcv9/libdll.so            sparc
604 usr/lib/lib-ldll
605 usr/lib/lib-ldll.ln
606 usr/lib/amd64/lib-ldll.ln            i386
607 usr/lib/sparcv9/lib-ldll.ln          sparc
608 # libpp (a helper library needed by AST's msgcc)
609 usr/lib/libpp.so
610 usr/lib/lib-lpp
611 usr/lib/lib-lpp.ln
612 usr/lib/locale/C/LC_MESSAGES/libpp
613 # libshell
614 usr/lib/libshell.so
615 usr/lib/amd64/libshell.so             i386
616 usr/lib/sparcv9/libshell.so          sparc
617 usr/lib/lib-lshell
618 usr/lib/lib-lshell.ln
619 usr/lib/amd64/lib-lshell.ln          i386
620 usr/lib/sparcv9/lib-lshell.ln        sparc
621 # libsum
622 usr/lib/libsum.so
623 usr/lib/amd64/libsum.so               i386
624 usr/lib/sparcv9/libsum.so            sparc
625 usr/lib/lib-lsum
626 usr/lib/lib-lsum.ln
627 usr/lib/amd64/lib-lsum.ln            i386
628 usr/lib/sparcv9/lib-lsum.ln          sparc
629 #
630 # This file is used in ON to build DSCP clients. It is not for customers.
631 #
632 usr/include/libdscp.h                 sparc
633 #
634 # These files are used by the iSCSI Target and the iSCSI Initiator
635 #
636 usr/include/sys/iscsi_protocol.h
637 usr/include/sys/iscsi_authclient.h
638 usr/include/sys/iscsi_authclientglue.h
639 #
640 # These files are used by the COMSTAR iSCSI target port provider
641 #
642 usr/include/sys/idm
643 usr/include/sys/iscsit/chap.h
644 usr/include/sys/iscsit/iscsi_if.h
645 usr/include/sys/iscsit/isns_protocol.h
646 usr/include/sys/iscsit/radius_packet.h
647 usr/include/sys/iscsit/radius_protocol.h
648 #
649 # libshare is private and the 64-bit sharemgr is not delivered.
650 #
651 usr/lib/libshare.so
652 usr/lib/amd64/libshare.so             i386
653 usr/lib/sparcv9/libshare.so          sparc
654 usr/lib/fs/autofs/libshare_autofs.so
655 usr/lib/fs/autofs/amd64/libshare_autofs.so i386

```

```

656 usr/lib/fs/autofs/sparcv9/libshare_autofs.so      sparc
657 usr/lib/fs/nfs/libshare_nfs.so
658 usr/lib/fs/nfs/amd64/libshare_nfs.so              i386
659 usr/lib/fs/nfs/sparcv9/libshare_nfs.so            sparc
660 usr/lib/fs/smb/libshare_smb.so
661 usr/lib/fs/smb/amd64/libshare_smb.so              i386
662 usr/lib/fs/smb/sparcv9/libshare_smb.so            sparc
663 usr/lib/fs/smbfs/libshare_smbfs.so
664 usr/lib/fs/smbfs/amd64/libshare_smbfs.so          i386
665 usr/lib/fs/smbfs/sparcv9/libshare_smbfs.so        sparc
666 usr/include/libshare_impl.h
667 usr/include/scfutil.h
668 #
669 # These files are installed in the proto area by the build of libpri for
670 # the benefit of the builds of FMA libldom, Zeus, picld plugins, and/or
671 # other libpri consumers. However, the libpri interfaces are private to
672 # Sun (Consolidation Private) and not intended for customer use. So these
673 # files (the symlink and the lint library) are excluded from packaging.
674 #
675 usr/lib/libpri.so                sparc
676 usr/lib/llib-lpri               sparc
677 usr/lib/llib-lpri.ln            sparc
678 usr/lib/sparcv9/libpri.so        sparc
679 usr/lib/sparcv9/llib-lpri.ln    sparc
680 #
681 # These files are installed in the proto area by the build of libds for
682 # the benefit of the builds of sun4v IO FMA and/or other libds
683 # consumers. However, the libds interfaces are private to Sun
684 # (Consolidation Private) and not intended for customer use. So these
685 # files (the symlink and the lint library) are excluded from packaging.
686 #
687 usr/lib/libds.so                sparc
688 usr/lib/sparcv9/libds.so        sparc
689 usr/lib/llib-lds               sparc
690 usr/lib/llib-lds.ln            sparc
691 usr/lib/sparcv9/llib-lds.ln    sparc
692 usr/lib/libdsconfig.so
693 usr/lib/llib-ldscfg.ln
694 usr/platform/sun4v/include/sys/libds.h    sparc
695 usr/platform/sun4v/include/sys/vlds.h     sparc
696 #
697 # Private/Internal u8_textprep header file. Do not ship.
698 #
699 usr/include/sys/u8_textprep_data.h
700 #
701 # SQLite is private, used by SMF (svc.configd), idmapd and libsmb.
702 #
703 usr/include/sqlite
704 usr/lib/libsqlite-native.o
705 usr/lib/libsqlite.o
706 usr/lib/llib-lsqlite.ln
707 usr/lib/smbdrv/libsqlite.so
708 #
709 # Private/Internal kiconv header files. Do not ship.
710 #
711 usr/include/sys/kiconv_big5_utf8.h
712 usr/include/sys/kiconv_ck_common.h
713 usr/include/sys/kiconv_cp950hkscs_utf8.h
714 usr/include/sys/kiconv_emeal.h
715 usr/include/sys/kiconv_emea2.h
716 usr/include/sys/kiconv_euckr_utf8.h
717 usr/include/sys/kiconv_euctw_utf8.h
718 usr/include/sys/kiconv_gb18030_utf8.h
719 usr/include/sys/kiconv_gb2312_utf8.h
720 usr/include/sys/kiconv_hkscs_utf8.h
721 usr/include/sys/kiconv_ja_jis_to_unicode.h

```

```

722 usr/include/sys/kiconv_ja_unicode_to_jis.h
723 usr/include/sys/kiconv_ja.h
724 usr/include/sys/kiconv_ko.h
725 usr/include/sys/kiconv_latin1.h
726 usr/include/sys/kiconv_sc.h
727 usr/include/sys/kiconv_tc.h
728 usr/include/sys/kiconv_uhc_utf8.h
729 usr/include/sys/kiconv_utf8_big5.h
730 usr/include/sys/kiconv_utf8_cp950hkscs.h
731 usr/include/sys/kiconv_utf8_euckr.h
732 usr/include/sys/kiconv_utf8_euctw.h
733 usr/include/sys/kiconv_utf8_gb18030.h
734 usr/include/sys/kiconv_utf8_gb2312.h
735 usr/include/sys/kiconv_utf8_hkscs.h
736 usr/include/sys/kiconv_utf8_uhc.h
737 #
738 # At this time, the ttydefs.cleanup file is only useful on sun4u systems
739 #
740 etc/flash/postdeployment/ttydefs.cleanup          i386
741 #
742 # This header file is shared only between the power commands and
743 # ppm/srn modules # and should not be in any package
744 #
745 usr/include/sys/srn.h
746 #
747 # Private/Internal header files of smbdrv. Do not ship.
748 #
749 usr/include/smb
750 usr/include/smbdrv
751 #
752 # Private/Internal dtrace scripts of smbdrv. Do not ship.
753 #
754 usr/lib/smbdrv/dtrace
755 #
756 # Private/Internal (lint) libraries of smbdrv. Do not ship.
757 #
758 usr/lib/reparse/llib-lreparse_smb
759 usr/lib/reparse/llib-lreparse_smb.ln
760 usr/lib/smbdrv/llib-lmlrpc
761 usr/lib/smbdrv/llib-lmlrpc.ln
762 usr/lib/smbdrv/llib-lmlsvc
763 usr/lib/smbdrv/llib-lmlsvc.ln
764 usr/lib/smbdrv/llib-lsmb
765 usr/lib/smbdrv/llib-lsmb.ln
766 usr/lib/smbdrv/llib-lsmbns
767 usr/lib/smbdrv/llib-lsmbns.ln
768 #
769 #
770 # Private/Internal 64-bit libraries of smbdrv. Do not ship.
771 #
772 usr/lib/smbdrv/amd64                i386
773 usr/lib/smbdrv/sparcv9              sparc
774 #
775 usr/lib/reparse/amd64/libreparse_smb.so          i386
776 usr/lib/reparse/amd64/libreparse_smb.so.1        i386
777 usr/lib/reparse/amd64/llib-lreparse_smb.ln       i386
778 usr/lib/reparse/sparcv9/libreparse_smb.so        sparc
779 usr/lib/reparse/sparcv9/libreparse_smb.so.1      sparc
780 usr/lib/reparse/sparcv9/llib-lreparse_smb.ln     sparc
781 #
782 # Private dirent, extended to include flags, for use by SMB server
783 #
784 usr/include/sys/extdirent.h
785 #
786 # Private header files for vsan service
787 #

```

```

788 usr/include/libvscan.h
789 usr/include/sys/vscan.h
790 #
791 # libvscan is private
792 #
793 usr/lib/vscan/llib-lvscan
794 usr/lib/vscan/llib-lvscan.ln
795 #
796 # i86hvm is not a full platform. It is just a home for paravirtualized
797 # drivers. There is no usr/ component to this sub-platform, but the
798 # directory is created in the proto area to keep other tools happy.
799 #
800 usr/platform/i86hvm                                i386
801 #
802 # Private sdcard framework headers
803 #
804 usr/include/sys/sdcard
805 #
806 # libsmbfs is private
807 #
808 usr/include/net smb
809 usr/lib/lib smbfs.so
810 usr/lib/amd64/lib smbfs.so                i386
811 usr/lib/sparcv9/lib smbfs.so              sparc
812 usr/lib/llib-lsmbfs
813 usr/lib/llib-lsmbfs.ln
814 usr/lib/amd64/llib-lsmbfs.ln            i386
815 usr/lib/sparcv9/llib-lsmbfs.ln          sparc
816 #
817 # demo & test program for smbfs (private) ACL support
818 #
819 usr/lib/fs/smbfs/chacl
820 usr/lib/fs/smbfs/lsacl
821 usr/lib/fs/smbfs/testnp
822 #
823 # FC related files
824 kernel/kmdb/fcip                                i386
825 kernel/kmdb/amd64/fcip                          i386
826 kernel/kmdb/sparcv9/fcip                        sparc
827 kernel/kmdb/fcp                                i386
828 kernel/kmdb/amd64/fcp                          i386
829 kernel/kmdb/sparcv9/fcp                        sparc
830 kernel/kmdb/fctl                               i386
831 kernel/kmdb/amd64/fctl                         i386
832 kernel/kmdb/sparcv9/fctl                      sparc
833 kernel/kmdb/qlc                                i386
834 kernel/kmdb/amd64/qlc                         i386
835 kernel/kmdb/sparcv9/qlc                      sparc
836 lib/llib-la5k                                  sparc
837 lib/llib-la5k.ln                              sparc
838 lib/sparcv9/llib-la5k.ln                      sparc
839 lib/llib-lg_fc                                sparc
840 lib/llib-lg_fc.ln                             sparc
841 lib/sparcv9/llib-lg_fc.ln                    sparc
842 usr/include/a_state.h                         sparc
843 usr/include/a5k.h                             sparc
844 usr/include/exec.h                            sparc
845 usr/include/g_scsi.h                          sparc
846 usr/include/g_state.h                        sparc
847 usr/include/gfc.h                            sparc
848 usr/include/l_common.h                       sparc
849 usr/include/l_error.h                       sparc
850 usr/include/rom.h                             sparc
851 usr/include/stgcom.h                         sparc
852 usr/include/sys/fibre-channel
853 usr/lib/llib-LHBAAPI

```

```

854 usr/lib/llib-LHBAAPI.ln
855 usr/lib/amd64/llib-LHBAAPI.ln            i386
856 usr/lib/sparcv9/llib-LHBAAPI.ln          sparc
857 #
858 usr/bin/dscfgcli
859 usr/bin/sd_diag
860 usr/bin/sd_stats
861 usr/include/nsctl.h
862 usr/include/sys/ncall
863 usr/include/sys/nsc_ddi.h
864 usr/include/sys/nsc_thread.h
865 usr/include/sys/nsctl
866 usr/include/sys/nskernd.h
867 usr/include/sys/unistat
868 usr/lib/libnsctl.so
869 usr/lib/librdc.so
870 usr/lib/libunistat.so
871 usr/lib/llib-lnsctl.ln
872 usr/lib/llib-lrdc.ln
873 usr/lib/llib-lunistat.ln
874 #
875 # These files are used by the iSCSI initiator only.
876 # No reason to ship them.
877 #
878 usr/include/sys/scsi/adapters/iscsi_door.h
879 usr/include/sys/scsi/adapters/iscsi_if.h
880 #
881 # sbd ioctl hdr
882 #
883 usr/include/sys/stmf_sbd_ioctl.h
884 #
885 # proxy port provider interface
886 #
887 usr/include/sys/pppt_ic_if.h
888 usr/include/sys/pppt_ioctl.h
889 #
890 # proxy daemon lint library
891 #
892 usr/lib/llib-lstmfproxy
893 usr/lib/llib-lstmfproxy.ln
894 usr/lib/amd64/llib-lstmfproxy.ln          i386
895 usr/lib/sparcv9/llib-lstmfproxy.ln        sparc
896 #
897 # portable object file and dictionary used by libfmd_msg test
898 #
899 usr/lib/fm/dict/TEST.dict
900 usr/lib/locale/C/LC_MESSAGES/TEST.mo
901 usr/lib/locale/C/LC_MESSAGES/TEST.po
902 #
903 # Private idmap RPC protocol
904 #
905 usr/include/rpcsvc/idmap_prot.h
906 usr/include/rpcsvc/idmap_prot.x
907 #
908 # Private idmap directory API
909 #
910 usr/include/directory.h
911 #
912 # librstp is private for bridging
913 #
914 usr/include/stp_bpdu.h
915 usr/include/stp_in.h
916 usr/include/stp_vectors.h
917 usr/lib/librstp.so
918 usr/lib/llib-lrstp
919 usr/lib/llib-lrstp.ln

```

```
920 #
921 # Private nvfru API
922 #
923 usr/include/nvfru.h
924 #
925 # vrrp
926 #
927 usr/include/libvrrpadm.h
928 usr/lib/libvrrpadm.so
929 usr/lib/amd64/libvrrpadm.so      i386
930 usr/lib/sparcv9/libvrrpadm.so    sparc
931 usr/lib/l1ib-lvrrpadm
932 usr/lib/l1ib-lvrrpadm.ln
933 usr/lib/amd64/l1ib-lvrrpadm.ln  i386
934 usr/lib/sparcv9/l1ib-lvrrpadm.ln sparc
935 #
936 # This is only used during the -t tools build
937 #
938 opt/onbld/bin/i386/elfsign      i386
939 opt/onbld/bin/sparc/elfsign     sparc
940
941 #
942 # Private libdwarf
943 #
944 opt/onbld/lib/i386/libdwarf.so  i386
945 opt/onbld/lib/sparc/libdwarf.so sparc
946
947 #
948 # Private socket filter API
949 #
950 usr/include/sys/sockfilter.h
951 #
952 # We don't actually validate license action payloads, and the license
953 # staging area is provided as a separate basedir for package
954 # publication. The net result is that everything therein should be
955 # ignored for packaging validation.
956 #
957 licenses
958 # Libbe is private
959 #
960 usr/include/libbe_priv.h
```



```
new/usr/src/cmd/cmd-inet/etc/sock2path.d/system%2Fkernel
*****
1303 Sat Aug 18 10:36:55 2012
new/usr/src/cmd/cmd-inet/etc/sock2path.d/system%2Fkernel
dccp: lint fixes, dccp_conn_create_v6
*****
1 # CDDL HEADER START
2 #
3 # The contents of this file are subject to the terms of the
4 # Common Development and Distribution License (the "License").
5 # You may not use this file except in compliance with the License.
6 #
7 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
8 # or http://www.opensolaris.org/os/licensing.
9 # See the License for the specific language governing permissions
10 # and limitations under the License.
11 #
12 # When distributing Covered Code, include this CDDL HEADER in each
13 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
14 # If applicable, add the following below this CDDL HEADER, with the
15 # fields enclosed by brackets "[]" replaced with your own identifying
16 # information: Portions Copyright [yyyy] [name of copyright owner]
17 #
18 # CDDL HEADER END
19 #
20 # Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
21 #
22 # socket configuration information
23 #
24 #      Family      Type  Protocol  Dev|Module
25 #          2          2      0      tcp
26 #          2          2      6      tcp
27
28 #      26          2      0      tcp
29 #      26          2      6      tcp
30
31 #          2          1      0      udp
32 #          2          1      17     udp
33
34 #      26          1      0      udp
35 #      26          1      17     udp
36
37 #          1          2      0      /dev/ticotsord
38 #          1          6      0      /dev/ticotsord
39 #          1          1      0      /dev/ticlts
40
41 #          2          4      0      icmp
42 #      26          4      0      icmp
43
44 #          2          2      132     socksctp
45 #      26          2      132     socksctp
46 #          2          6      132     socksctp
47 #      26          6      132     socksctp
48
49 #      24          4      0      rts
50
51 #      27          4      2      /dev/keysock
52 #      29          4      1      /dev/spdsock
53
54 #      31          1      0      trill
55
56 #          2          6      33      dccp
57 #endif /* ! codereview */
```

```

*****
189148 Sat Aug 18 10:36:55 2012
new/usr/src/cmd/cmd-inet/usr.bin/netstat/netstat.c
dccc: complete netstack
*****
_____unchanged_portion_omitted_____

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

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

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

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

```

```

197 static void             dccc_report(const mib_item_t *item);
198 #endif /* ! codereview */
199 static void             print_ip_stats(mib2_ip_t *ip);
200 static void             print_icmp_stats(mib2_icmp_t *icmp);
201 static void             print_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6);
202 static void             print_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6);
203 static void             print_sctp_stats(mib2_sctp_t *tcp);
204 static void             print_tcp_stats(mib2_tcp_t *tcp);
205 static void             print_udp_stats(mib2_udp_t *udp);
206 static void             print_rawip_stats(mib2_rawip_t *rawip);
207 static void             print_igmp_stats(struct igmpstat *igps);
208 static void             print_mrt_stats(struct mrtstat *mrts);
209 static void             print_dccc_stats(mib2_dccc_t *dccc);
196 static void             sctp_report(const mib_item_t *item);
210 static void             sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6,
211         mib2_ipv6IfStatsEntry_t *sum6);
212 static void             sum_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6,
213         mib2_ipv6IfIcmpEntry_t *sum6);
214 static void             m_report(void);
215 static void             dhcp_report(char *);

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

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

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

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

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

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

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

```

```

262 */
263 static int ipAddrEntrySize;
264 static int ipRouteEntrySize;
265 static int ipNetToMediaEntrySize;
266 static int ipMemberEntrySize;
267 static int ipGroupSourceEntrySize;
268 static int ipRouteAttributeSize;
269 static int vifctlSize;
270 static int mfctlSize;

272 static int ipv6IfStatsEntrySize;
273 static int ipv6IfIcmpEntrySize;
274 static int ipv6AddrEntrySize;
275 static int ipv6RouteEntrySize;
276 static int ipv6NetToMediaEntrySize;
277 static int ipv6MemberEntrySize;
278 static int ipv6GroupSourceEntrySize;

280 static int ipDestEntrySize;

282 static int transportMLPSize;
283 static int tcpConnEntrySize;
284 static int tcp6ConnEntrySize;
285 static int udpEntrySize;
286 static int udp6EntrySize;
287 static int sctpEntrySize;
288 static int sctpLocalEntrySize;
289 static int sctpRemoteEntrySize;
290 static int dccpEntrySize;
291 static int dccp6EntrySize;
292 #endif /* ! codereview */

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

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

299 static const char *filter_keys[NFILTERKEYS] = {
300     "af", "outif", "dst", "flags"
301 };

303 static m_label_t *zone_security_label = NULL;

305 /* Flags on routes */
306 #define FLF_A      0x00000001
307 #define FLF_b      0x00000002
308 #define FLF_D      0x00000004
309 #define FLF_G      0x00000008
310 #define FLF_H      0x00000010
311 #define FLF_L      0x00000020
312 #define FLF_U      0x00000040
313 #define FLF_M      0x00000080
314 #define FLF_S      0x00000100
315 #define FLF_C      0x00000200 /* IRE_IF_CLONE */
316 #define FLF_I      0x00000400 /* RTF_INDIRECT */
317 #define FLF_R      0x00000800 /* RTF_REJECT */
318 #define FLF_B      0x00001000 /* RTF_BLACKHOLE */
319 #define FLF_Z      0x00100000 /* RTF_ZONE */

321 static const char flag_list[] = "AbdGHLUMSCIRBZ";

323 typedef struct filter_rule filter_t;

325 struct filter_rule {
326     filter_t *f_next;
327     union {

```

```

328     int f_family;
329     const char *f_ifname;
330     struct {
331         struct hostent *f_address;
332         in6_addr_t f_mask;
333     } a;
334     struct {
335         uint_t f_flagset;
336         uint_t f_flagclear;
337     } f;
338 } u;
339 };

341 /*
342  * The user-specified filters are linked into lists separated by
343  * keyword (type of filter). Thus, the matching algorithm is:
344  * For each non-empty filter list
345  *     If no filters in the list match
346  *         then stop here; route doesn't match
347  * If loop above completes, then route does match and will be
348  * displayed.
349 */
350 static filter_t *filters[NFILTERKEYS];

352 static uint_t timestamp_fmt = NODATE;

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

358 int
359 main(int argc, char **argv)
360 {
361     char *name;
362     mib_item_t *item = NULL;
363     mib_item_t *previtem = NULL;
364     int sd = -1;
365     char *ifname = NULL;
366     int interval = 0; /* Single time by default */
367     int count = -1; /* Forever */
368     int c;
369     int d;
370     /*
371      * Possible values of 'iflag_only':
372      * -1, no feature-flags;
373      * 0, IFlag and other feature-flags enabled
374      * 1, IFlag is the only feature-flag enabled
375      * : trinary variable, modified using IFLAGMOD()
376      */
377     int iflag_only = -1;
378     boolean_t once_only = B_FALSE; /* '-i' with count > 1 */
379     extern char *optarg;
380     extern int optind;
381     char *default_ip_str = NULL;

383     name = argv[0];

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

391     (void) setlocale(LC_ALL, "");
392     (void) textdomain(TEXT_DOMAIN);

```

```

394 while ((c = getopt(argc, argv, "adimnrspMgvxf:P:I:DRT:")) != -1) {
395     switch ((char)c) {
396         case 'a': /* all connections */
397             Aflag = B_TRUE;
398             break;
400         case 'd': /* DCE info */
401             Dflag = B_TRUE;
402             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
403             break;
405         case 'i': /* interface (ill/ipif report) */
406             Iflag = B_TRUE;
407             IFLAGMOD(Iflag_only, -1, 1); /* '-i' exists */
408             break;
410         case 'm': /* streams msg report */
411             Mflag = B_TRUE;
412             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
413             break;
415         case 'n': /* numeric format */
416             Nflag = B_TRUE;
417             break;
419         case 'r': /* route tables */
420             Rflag = B_TRUE;
421             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
422             break;
424         case 'R': /* security attributes */
425             RSECflag = B_TRUE;
426             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
427             break;
429         case 's': /* per-protocol statistics */
430             Sflag = B_TRUE;
431             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
432             break;
434         case 'p': /* arp/ndp table */
435             Pflag = B_TRUE;
436             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
437             break;
439         case 'M': /* multicast routing tables */
440             MMflag = B_TRUE;
441             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
442             break;
444         case 'g': /* multicast group membership */
445             Gflag = B_TRUE;
446             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
447             break;
449         case 'v': /* verbose output format */
450             Vflag = B_TRUE;
451             IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
452             break;
454         case 'x': /* turn on debugging */
455             Xflag = B_TRUE;
456             break;
458         case 'f':
459             process_filter(optarg);

```

```

460         break;
462         case 'P':
463             if (strcmp(optarg, "ip") == 0) {
464                 proto = IPPROTO_IP;
465             } else if (strcmp(optarg, "ipv6") == 0 ||
466                 strcmp(optarg, "ip6") == 0) {
467                 v4compat = 0; /* Overridden */
468                 proto = IPPROTO_IPV6;
469             } else if (strcmp(optarg, "icmp") == 0) {
470                 proto = IPPROTO_ICMP;
471             } else if (strcmp(optarg, "icmpv6") == 0 ||
472                 strcmp(optarg, "icmp6") == 0) {
473                 v4compat = 0; /* Overridden */
474                 proto = IPPROTO_ICMPV6;
475             } else if (strcmp(optarg, "igmp") == 0) {
476                 proto = IPPROTO_IGMP;
477             } else if (strcmp(optarg, "udp") == 0) {
478                 proto = IPPROTO_UDP;
479             } else if (strcmp(optarg, "tcp") == 0) {
480                 proto = IPPROTO_TCP;
481             } else if (strcmp(optarg, "sctp") == 0) {
482                 proto = IPPROTO_SCTP;
483             } else if (strcmp(optarg, "raw") == 0 ||
484                 strcmp(optarg, "rawip") == 0) {
485                 proto = IPPROTO_RAW;
486             } else if (strcmp(optarg, "dccp") == 0) {
487                 proto = IPPROTO_DCCP;
488             #endif /* ! codereview */
489             } else {
490                 fatal(1, "%s: unknown protocol.\n", optarg);
491             }
492             break;
494         case 'I':
495             ifname = optarg;
496             Iflag = B_TRUE;
497             IFLAGMOD(Iflag_only, -1, 1); /* see macro def'n */
498             break;
500         case 'D':
501             DHCPflag = B_TRUE;
502             Iflag_only = 0;
503             break;
505         case 'T':
506             if (optarg) {
507                 if (*optarg == 'u')
508                     timestamp_fmt = UDATE;
509                 else if (*optarg == 'd')
510                     timestamp_fmt = DDATE;
511                 else
512                     usage(name);
513             } else {
514                 usage(name);
515             }
516             break;
518         case '?:
519         default:
520             usage(name);
521         }
522     }
524     /*
525     * Make sure -R option is set only on a labeled system.

```

```

526     */
527     if (RSECflag && !is_system_labeled()) {
528         (void) fprintf(stderr, "-R set but labeling is not enabled\n");
529         usage(name);
530     }
531
532     /*
533     * Handle other arguments: find interval, count; the
534     * flags that accept 'interval' and 'count' are OR'd
535     * in the outermost 'if'; more flags may be added as
536     * required
537     */
538     if (Iflag || Sflag || Mflag) {
539         for (d = optind; d < argc; d++) {
540             if (isnum(argv[d])) {
541                 interval = atoi(argv[d]);
542                 if (d + 1 < argc &&
543                     isnum(argv[d + 1])) {
544                     count = atoi(argv[d + 1]);
545                     optind++;
546                 }
547                 optind++;
548                 if (interval == 0 || count == 0)
549                     usage(name);
550                 break;
551             }
552         }
553     }
554     if (optind < argc) {
555         if (Iflag && isnum(argv[optind])) {
556             count = atoi(argv[optind]);
557             if (count == 0)
558                 usage(name);
559             optind++;
560         }
561     }
562     if (optind < argc) {
563         (void) fprintf(stderr,
564             "%s: extra arguments\n", name);
565         usage(name);
566     }
567     if (interval)
568         setbuf(stdout, NULL);
569
570     if (DHCPflag) {
571         dhcp_report(Iflag ? ifname : NULL);
572         exit(0);
573     }
574
575     /*
576     * Get this process's security label if the -R switch is set.
577     * We use this label as the current zone's security label.
578     */
579     if (RSECflag) {
580         zone_security_label = m_label_alloc(MAC_LABEL);
581         if (zone_security_label == NULL)
582             fatal(errno, "m_label_alloc() failed");
583         if (getlabel(zone_security_label) < 0)
584             fatal(errno, "getlabel() failed");
585     }
586
587     /* Get data structures: priming before iteration */
588     if (family_selected(AF_INET) || family_selected(AF_INET6)) {
589         sd = mibopen();
590         if (sd == -1)
591             fatal(1, "can't open mib stream\n");

```

```

592         if ((item = mibget(sd)) == NULL) {
593             (void) close(sd);
594             fatal(1, "mibget() failed\n");
595         }
596         /* Extract constant sizes - need do once only */
597         mib_get_constants(item);
598     }
599     if ((kc = kstat_open()) == NULL) {
600         mibfree(item);
601         (void) close(sd);
602         fail(1, "kstat_open(): can't open /dev/kstat");
603     }
604
605     if (interval <= 0) {
606         count = 1;
607         once_only = B_TRUE;
608     }
609     /* 'for' loop 1: */
610     for (;;) {
611         mib_item_t *curritem = NULL; /* only for -[M]s */
612
613         if (timestamp_fmt != NODATE)
614             print_timestamp(timestamp_fmt);
615
616         /* netstat: AF_INET[6] behaviour */
617         if (family_selected(AF_INET) || family_selected(AF_INET6)) {
618             if (Sflag) {
619                 curritem = mib_item_diff(previtem, item);
620                 if (curritem == NULL)
621                     fatal(1, "can't process mib data, "
622                         "out of memory\n");
623                 mib_item_destroy(&previtem);
624             }
625
626             if (!Dflag || Iflag || Rflag || Sflag || Mflag ||
627                 MMflag || Pflag || Gflag || DHCPflag) {
628                 if (protocol_selected(IPPROTO_UDP))
629                     udp_report(item);
630                 if (protocol_selected(IPPROTO_TCP))
631                     tcp_report(item);
632                 if (protocol_selected(IPPROTO_SCTP))
633                     sctp_report(item);
634                 if (protocol_selected(IPPROTO_DCCP))
635                     dccp_report(item);
636             }
637             #endif /* ! codereview */
638             if (Iflag)
639                 if_report(item, ifname, Iflag_only, once_only);
640             if (Mflag)
641                 m_report();
642             if (Rflag)
643                 ire_report(item);
644             if (Sflag && MMflag) {
645                 mrt_stat_report(curritem);
646             } else {
647                 if (Sflag)
648                     stat_report(curritem);
649                 if (MMflag)
650                     mrt_report(item);
651             }
652             if (Gflag)
653                 group_report(item);
654             if (Pflag) {
655                 if (family_selected(AF_INET))
656                     arp_report(item);
657                 if (family_selected(AF_INET6))

```

```

658         ndp_report(item);
659     }
660     if (Dflag)
661         dce_report(item);
662     mib_item_destroy(&curritem);
663 }
664
665 /* netstat: AF_UNIX behaviour */
666 if (family_selected(AF_UNIX) &&
667     !(Dflag || Iflag || Rflag || Sflag || Mflag ||
668        MMflag || Pflag || Gflag))
669     unixpr(kc);
670 (void) kstat_close(kc);
671
672 /* iteration handling code */
673 if (count > 0 && --count == 0)
674     break;
675 (void) sleep(interval);
676
677 /* re-populating of data structures */
678 if (family_selected(AF_INET) || family_selected(AF_INET6)) {
679     if (Sflag) {
680         /* previtem is a cut-down list */
681         previtem = mib_item_dup(item);
682         if (previtem == NULL)
683             fatal(1, "can't process mib data, "
684                  "out of memory\n");
685     }
686     mibfree(item);
687     (void) close(sd);
688     if ((sd = mibopen()) == -1)
689         fatal(1, "can't open mib stream anymore\n");
690     if ((item = mibget(sd)) == NULL) {
691         (void) close(sd);
692         fatal(1, "mibget() failed\n");
693     }
694 }
695 if ((kc = kstat_open()) == NULL)
696     fail(1, "kstat_open(): can't open /dev/kstat");
697
698 } /* 'for' loop 1 ends */
699 mibfree(item);
700 (void) close(sd);
701 if (zone_security_label != NULL)
702     m_label_free(zone_security_label);
703
704 return (0);
705 }
706
707 static int
708 isnum(char *p)
709 {
710     int    len;
711     int    i;
712
713     len = strlen(p);
714     for (i = 0; i < len; i++)
715         if (!isdigit(p[i]))
716             return (0);
717     return (1);
718 }
719
720 /* ----- MIBGET ----- */

```

```

724 static mib_item_t *
725 mibget(int sd)
726 {
727     /*
728      * buf is an automatic for this function, so the
729      * compiler has complete control over its alignment;
730      * it is assumed this alignment is satisfactory for
731      * it to be casted to certain other struct pointers
732      * here, such as struct T_optmgmt_ack *.
733      */
734     uintptr_t    buf[512 / sizeof (uintptr_t)];
735     int          flags;
736     int          i, j, getcode;
737     struct strbuf    ctlbuf, databuf;
738     struct T_optmgmt_req    *tor = (struct T_optmgmt_req *)buf;
739     struct T_optmgmt_ack    *toa = (struct T_optmgmt_ack *)buf;
740     struct T_error_ack    *tea = (struct T_error_ack *)buf;
741     struct opthdr    *req;
742     mib_item_t    *first_item = NULL;
743     mib_item_t    *last_item = NULL;
744     mib_item_t    *temp;
745
746     tor->PRIM_type = T_SVR4_OPTMGMT_REQ;
747     tor->OPT_offset = sizeof (struct T_optmgmt_req);
748     tor->OPT_length = sizeof (struct opthdr);
749     tor->MGMT_flags = T_CURRENT;
750
751     /*
752      * Note: we use the special level value below so that IP will return
753      * us information concerning IRE_MARK_TESTHIDDEN routes.
754      */
755     req = (struct opthdr *)&tor[1];
756     req->level = EXPER_IP_AND_ALL_IRES;
757     req->name = 0;
758     req->len = 1;
759
760     ctlbuf.buf = (char *)buf;
761     ctlbuf.len = tor->OPT_length + tor->OPT_offset;
762     flags = 0;
763     if (putmsg(sd, &ctlbuf, (struct strbuf *)0, flags) == -1) {
764         perror("mibget: putmsg(ctl) failed");
765         goto error_exit;
766     }
767
768     /*
769      * Each reply consists of a ctl part for one fixed structure
770      * or table, as defined in mib2.h. The format is a T_OPTMGMT_ACK,
771      * containing an opthdr structure. level/name identify the entry,
772      * len is the size of the data part of the message.
773      */
774     req = (struct opthdr *)&toa[1];
775     ctlbuf.maxlen = sizeof (buf);
776     j = 1;
777     for (;;) {
778         flags = 0;
779         getcode = getmsg(sd, &ctlbuf, (struct strbuf *)0, &flags);
780         if (getcode == -1) {
781             perror("mibget getmsg(ctl) failed");
782             if (Xflag) {
783                 (void) fputs("# level name len\n",
784                             stderr);
785                 i = 0;
786                 for (last_item = first_item; last_item;
787                     last_item = last_item->next_item)
788                     (void) printf("%d %4d %5d %d\n",
789                                 i, req->level, req->name, req->len);
790             }
791             return (NULL);
792         }
793         if (Xflag) {
794             (void) fputs("# level name len\n",
795                         stderr);
796             i = 0;
797             for (last_item = first_item; last_item;
798                 last_item = last_item->next_item)
799                 (void) printf("%d %4d %5d %d\n",
800                             i, req->level, req->name, req->len);
801         }
802         temp = mib_item_dup(req);
803         if (temp == NULL)
804             return (NULL);
805         if (first_item == NULL)
806             first_item = temp;
807         last_item = temp;
808         temp = NULL;
809     }
810     return (first_item);
811 }

```

```

790         ++i,
791         last_item->group,
792         last_item->mib_id,
793         last_item->length);
794     }
795     goto error_exit;
796 }
797 if (getcode == 0 &&
798     ctlbuf.len >= sizeof (struct T_optmgmt_ack) &&
799     toa->PRIM_type == T_OPTMGMT_ACK &&
800     toa->MGMT_flags == T_SUCCESS &&
801     req->len == 0) {
802     if (Xflag)
803         (void) printf("mibget getmsg() %d returned "
804             "EOD (level %ld, name %ld)\n",
805             j, req->level, req->name);
806     return (first_item); /* this is EOD msg */
807 }
808
809 if (ctlbuf.len >= sizeof (struct T_error_ack) &&
810     tea->PRIM_type == T_ERROR_ACK) {
811     (void) fprintf(stderr,
812         "mibget %d gives T_ERROR_ACK: TLI_error = 0x%lx, "
813         "UNIX_error = 0x%lx\n",
814         j, tea->TLI_error, tea->UNIX_error);
815
816     errno = (tea->TLI_error == TSYSEERR) ?
817         tea->UNIX_error : EPROTO;
818     goto error_exit;
819 }
820
821 if (getcode != MOREDATA ||
822     ctlbuf.len < sizeof (struct T_optmgmt_ack) ||
823     toa->PRIM_type != T_OPTMGMT_ACK ||
824     toa->MGMT_flags != T_SUCCESS) {
825     (void) printf("mibget getmsg(ctl) %d returned %d, "
826         "ctlbuf.len = %d, PRIM_type = %ld\n",
827         j, getcode, ctlbuf.len, toa->PRIM_type);
828
829     if (toa->PRIM_type == T_OPTMGMT_ACK)
830         (void) printf("T_OPTMGMT_ACK: "
831             "MGMT_flags = 0x%lx, req->len = %ld\n",
832             toa->MGMT_flags, req->len);
833     errno = ENOMSG;
834     goto error_exit;
835 }
836
837 temp = (mib_item_t *)malloc(sizeof (mib_item_t));
838 if (temp == NULL) {
839     perror("mibget malloc failed");
840     goto error_exit;
841 }
842 if (last_item != NULL)
843     last_item->next_item = temp;
844 else
845     first_item = temp;
846 last_item = temp;
847 last_item->next_item = NULL;
848 last_item->group = req->level;
849 last_item->mib_id = req->name;
850 last_item->length = req->len;
851 last_item->valp = malloc((int)req->len);
852 if (last_item->valp == NULL)
853     goto error_exit;
854 if (Xflag)
855     (void) printf("msg %d: group = %4d   mib_id = %5d"

```

```

856         "length = %d\n",
857         j, last_item->group, last_item->mib_id,
858         last_item->length);
859
860     databuf.maxlen = last_item->length;
861     databuf.buf = (char *)last_item->valp;
862     databuf.len = 0;
863     flags = 0;
864     getcode = getmsg(sd, (struct strbuf *)0, &databuf, &flags);
865     if (getcode == -1) {
866         perror("mibget getmsg(data) failed");
867         goto error_exit;
868     } else if (getcode != 0) {
869         (void) printf("mibget getmsg(data) returned %d, "
870             "databuf.maxlen = %d, databuf.len = %d\n",
871             getcode, databuf.maxlen, databuf.len);
872         goto error_exit;
873     }
874     j++;
875 }
876 /* NOTREACHED */
877
878 error_exit:;
879     mibfree(first_item);
880     return (NULL);
881 }
882
883 /*
884  * mibfree: frees a linked list of type (mib_item_t *)
885  * returned by mibget(); this is NOT THE SAME AS
886  * mib_item_destroy(), so should be used for objects
887  * returned by mibget() only
888  */
889 static void
890 mibfree(mib_item_t *firstitem)
891 {
892     mib_item_t *lastitem;
893
894     while (firstitem != NULL) {
895         lastitem = firstitem;
896         firstitem = firstitem->next_item;
897         if (lastitem->valp != NULL)
898             free(lastitem->valp);
899         free(lastitem);
900     }
901 }
902
903 static int
904 mibopen(void)
905 {
906     int     sd;
907
908     sd = open("/dev/arp", O_RDWR);
909     if (sd == -1) {
910         perror("arp open");
911         return (-1);
912     }
913     if (ioctl(sd, I_PUSH, "tcp") == -1) {
914         perror("tcp I_PUSH");
915         (void) close(sd);
916         return (-1);
917     }
918     if (ioctl(sd, I_PUSH, "udp") == -1) {
919         perror("udp I_PUSH");
920         (void) close(sd);
921         return (-1);

```

```

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

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

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

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

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

```

```

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

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

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

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

```



```

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

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

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

```

```

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

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

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

```

```

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

```

```

1252         MDIFF(d, i2, i1, icmpOutMsgs);
1253         MDIFF(d, i2, i1, icmpOutDrops);
1254         MDIFF(d, i2, i1, icmpOutErrors);
1255         MDIFF(d, i2, i1, icmpOutDestUnreachs);
1256         MDIFF(d, i2, i1, icmpOutTimeExcds);
1257         MDIFF(d, i2, i1, icmpOutParmProbs);
1258         MDIFF(d, i2, i1, icmpOutSrcQuenchs);
1259         MDIFF(d, i2, i1, icmpOutRedirects);
1260         MDIFF(d, i2, i1, icmpOutEchos);
1261         MDIFF(d, i2, i1, icmpOutEchoReps);
1262         MDIFF(d, i2, i1, icmpOutTimestamps);
1263         MDIFF(d, i2, i1, icmpOutTimestampReps);
1264         MDIFF(d, i2, i1, icmpOutAddrMaskReps);
1265         MDIFF(d, i2, i1, icmpOutFragNeeded);
1266         MDIFF(d, i2, i1, icmpInOverflows);
1267         prevp = diffptr++;
1268         break;
1269     }
1270     case MIB2_ICMP6: {
1271         mib2_ip6IfIcmpEntry_t *i2;
1272         mib2_ip6IfIcmpEntry_t *i1;
1273         mib2_ip6IfIcmpEntry_t *d;
1274
1275         i2 = (mib2_ip6IfIcmpEntry_t *)tempp2->valp;
1276         i1 = (mib2_ip6IfIcmpEntry_t *)tempp1->valp;
1277         diffptr->group = tempp2->group;
1278         diffptr->mib_id = tempp2->mib_id;
1279         diffptr->length = tempp2->length;
1280         d = (mib2_ip6IfIcmpEntry_t *)calloc(tempp2->length, 1);
1281         if (d == NULL)
1282             goto mibdiff_out_of_memory;
1283         diffptr->valp = d;
1284         MDIFF(d, i2, i1, ipv6IfIcmpInMsgs);
1285         MDIFF(d, i2, i1, ipv6IfIcmpInErrors);
1286         MDIFF(d, i2, i1, ipv6IfIcmpInDestUnreachs);
1287         MDIFF(d, i2, i1, ipv6IfIcmpInAdminProhibs);
1288         MDIFF(d, i2, i1, ipv6IfIcmpInTimeExcds);
1289         MDIFF(d, i2, i1, ipv6IfIcmpInParmProblems);
1290         MDIFF(d, i2, i1, ipv6IfIcmpInPktTooBigs);
1291         MDIFF(d, i2, i1, ipv6IfIcmpInEchos);
1292         MDIFF(d, i2, i1, ipv6IfIcmpInEchoReplies);
1293         MDIFF(d, i2, i1, ipv6IfIcmpInRouterSolicits);
1294         MDIFF(d, i2, i1, ipv6IfIcmpInRouterAdvertisements);
1295         MDIFF(d, i2, i1, ipv6IfIcmpInNeighborSolicits);
1296         MDIFF(d, i2, i1, ipv6IfIcmpInNeighborAdvertisements);
1297         MDIFF(d, i2, i1, ipv6IfIcmpInRedirects);
1298         MDIFF(d, i2, i1, ipv6IfIcmpInBadRedirects);
1299         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembQueries);
1300         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembResponses);
1301         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembReductions);
1302         MDIFF(d, i2, i1, ipv6IfIcmpInOverflows);
1303         MDIFF(d, i2, i1, ipv6IfIcmpOutMsgs);
1304         MDIFF(d, i2, i1, ipv6IfIcmpOutErrors);
1305         MDIFF(d, i2, i1, ipv6IfIcmpOutDestUnreachs);
1306         MDIFF(d, i2, i1, ipv6IfIcmpOutAdminProhibs);
1307         MDIFF(d, i2, i1, ipv6IfIcmpOutTimeExcds);
1308         MDIFF(d, i2, i1, ipv6IfIcmpOutParmProblems);
1309         MDIFF(d, i2, i1, ipv6IfIcmpOutPktTooBigs);
1310         MDIFF(d, i2, i1, ipv6IfIcmpOutEchos);
1311         MDIFF(d, i2, i1, ipv6IfIcmpOutEchoReplies);
1312         MDIFF(d, i2, i1, ipv6IfIcmpOutRouterSolicits);
1313         MDIFF(d, i2, i1, ipv6IfIcmpOutRouterAdvertisements);
1314         MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborSolicits);
1315         MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborAdvertisements);
1316         MDIFF(d, i2, i1, ipv6IfIcmpOutRedirects);
1317     }

```

```

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

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

```

```

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

1396         u2 = (mib2_udp_t *)tempp2->valp;
1397         u1 = (mib2_udp_t *)tempp1->valp;
1398         diffptr->group = tempp2->group;
1399         diffptr->mib_id = tempp2->mib_id;
1400         diffptr->length = tempp2->length;
1401         d = (mib2_udp_t *)calloc(tempp2->length, 1);
1402         if (d == NULL)
1403             goto mibdiff_out_of_memory;
1404         diffptr->valp = d;
1405         MDIFF(d, u2, u1, udpHCInDatagrams);
1406         MDIFF(d, u2, u1, udpInErrors);
1407         MDIFF(d, u2, u1, udpHCOutDatagrams);
1408         MDIFF(d, u2, u1, udpOutErrors);
1409         prevp = diffptr++;
1410         break;
1411     }
1412     case MIB2 SCTP: {
1413         mib2_sctp_t *s2;
1414         mib2_sctp_t *s1;
1415         mib2_sctp_t *d;

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

```

```

1450         MDIFF(d, s2, s1, sctpInOrderChunks);
1451         MDIFF(d, s2, s1, sctpInUnorderChunks);
1452         MDIFF(d, s2, s1, sctpInAck);
1453         MDIFF(d, s2, s1, sctpInDupAck);
1454         MDIFF(d, s2, s1, sctpInAckUnsent);
1455         MDIFF(d, s2, s1, sctpFragUsrMsgs);
1456         MDIFF(d, s2, s1, sctpReasmUsrMsgs);
1457         MDIFF(d, s2, s1, sctpOutSCTPPkts);
1458         MDIFF(d, s2, s1, sctpInSCTPPkts);
1459         MDIFF(d, s2, s1, sctpInInvalidCookie);
1460         MDIFF(d, s2, s1, sctpTimRetrans);
1461         MDIFF(d, s2, s1, sctpTimRetransDrop);
1462         MDIFF(d, s2, s1, sctpTimHeartBeatProbe);
1463         MDIFF(d, s2, s1, sctpTimHeartBeatDrop);
1464         MDIFF(d, s2, s1, sctpListenDrop);
1465         MDIFF(d, s2, s1, sctpInClosed);
1466         prevp = diffptr++;
1467         break;
1468     }
1469     case MIB2_DCCP: {
1470         /* XXX:DCCP */
1471         break;
1472     }
1473 #endif /* ! codereview */
1474     case EXPER_RAWIP: {
1475         mib2_rawip_t *r2;
1476         mib2_rawip_t *r1;
1477         mib2_rawip_t *d;
1478
1479         r2 = (mib2_rawip_t *)tempp2->valp;
1480         r1 = (mib2_rawip_t *)tempp1->valp;
1481         diffptr->group = tempp2->group;
1482         diffptr->mib_id = tempp2->mib_id;
1483         diffptr->length = tempp2->length;
1484         d = (mib2_rawip_t *)calloc(tempp2->length, 1);
1485         if (d == NULL)
1486             goto mibdiff_out_of_memory;
1487         diffptr->valp = d;
1488         MDIFF(d, r2, r1, rawipInDatagrams);
1489         MDIFF(d, r2, r1, rawipInErrors);
1490         MDIFF(d, r2, r1, rawipInChecksumErrs);
1491         MDIFF(d, r2, r1, rawipOutDatagrams);
1492         MDIFF(d, r2, r1, rawipOutErrors);
1493         prevp = diffptr++;
1494         break;
1495     }
1496     /*
1497     * there are more "group" types but they aren't
1498     * required for the -s and -Ms options
1499     */
1500 } /* 'for' loop 2 ends */
1501 tempp1 = NULL;
1502 } /* 'for' loop 1 ends */
1503 tempp2 = NULL;
1504 diffptr--;
1505 diffptr->next_item = NULL;
1506 return (diffp);
1507
1508 mibdiff_out_of_memory:
1509     mib_item_destroy(&diffp);
1510     return (NULL);
1511 }
1512
1513 /*
1514 * mib_item_destroy: cleans up a mib_item_t *

```

```

1516 * that was created by calling mib_item_dup or
1517 * mib_item_diff
1518 */
1519 static void
1520 mib_item_destroy(mib_item_t **itemp) {
1521     int     nitems = 0;
1522     int     c = 0;
1523     mib_item_t *tempp;
1524
1525     if (itemp == NULL || *itemp == NULL)
1526         return;
1527
1528     for (tempp = *itemp; tempp != NULL; tempp = tempp->next_item)
1529         if (tempp->mib_id == 0)
1530             nitems++;
1531     else
1532         return; /* cannot destroy! */
1533
1534     if (nitems == 0)
1535         return; /* cannot destroy! */
1536
1537     for (c = nitems - 1; c >= 0; c--) {
1538         if ((itemp[0][c]).valp != NULL)
1539             free((itemp[0][c]).valp);
1540     }
1541     free(*itemp);
1542
1543     *itemp = NULL;
1544 }
1545
1546 /* Compare two Octet_ts. Return B_TRUE if they match, B_FALSE if not. */
1547 static boolean_t
1548 octetstrmatch(const Octet_t *a, const Octet_t *b)
1549 {
1550     if (a == NULL || b == NULL)
1551         return (B_FALSE);
1552
1553     if (a->o_length != b->o_length)
1554         return (B_FALSE);
1555
1556     return (memcmp(a->o_bytes, b->o_bytes, a->o_length) == 0);
1557 }
1558
1559 /* If octetstr() changes make an appropriate change to STR_EXPAND */
1560 static char *
1561 octetstr(const Octet_t *op, int code, char *dst, uint_t dstlen)
1562 {
1563     int     i;
1564     char    *cp;
1565
1566     cp = dst;
1567     if (op) {
1568         for (i = 0; i < op->o_length; i++) {
1569             switch (code) {
1570                 case 'd':
1571                     if (cp - dst + 4 > dstlen) {
1572                         *cp = '\0';
1573                         return (dst);
1574                     }
1575                     (void) snprintf(cp, 5, "%d.",
1576                                     0xff & op->o_bytes[i]);
1577                     cp = strchr(cp, '\0');
1578                     break;
1579                 case 'a':
1580                     if (cp - dst + 1 > dstlen) {
1581                         *cp = '\0';

```

```

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

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

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

```

```

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

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

1669 return (cp);
1670 }

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

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

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

1706 return (cp);
1707 }

1709 static int odd;

1711 static void
1712 prval_init(void)
1713 {

```

```

1714         odd = 0;
1715     }

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

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

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

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

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

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

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

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

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

```

```

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

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

1809         vifctlSize = mrts->mrts_vifctlSize;
1810         mfcctlSize = mrts->mrts_mfcctlSize;
1811         assert(IS_P2ALIGNED(vifctlSize,
1812             sizeof (struct vifctl *)));
1813         assert(IS_P2ALIGNED(mfcctlSize,
1814             sizeof (struct mfcctl *)));
1815         break;
1816     }
1817     case MIB2_IP6: {
1818         mib2_ip6IfStatsEntry_t *ip6;
1819         /* Just use the first entry */

1821         ip6 = (mib2_ip6IfStatsEntry_t *)item->valp;
1822         ipv6IfStatsEntrySize = ip6->ipv6IfStatsEntrySize;
1823         ipv6AddrEntrySize = ip6->ipv6AddrEntrySize;
1824         ipv6RouteEntrySize = ip6->ipv6RouteEntrySize;
1825         ipv6NetToMediaEntrySize = ip6->ipv6NetToMediaEntrySize;
1826         ipv6MemberEntrySize = ip6->ipv6MemberEntrySize;
1827         ipv6GroupSourceEntrySize =
1828             ip6->ipv6GroupSourceEntrySize;
1829         assert(IS_P2ALIGNED(ipv6IfStatsEntrySize,
1830             sizeof (mib2_ip6IfStatsEntry_t *)));
1831         assert(IS_P2ALIGNED(ipv6AddrEntrySize,
1832             sizeof (mib2_ip6AddrEntry_t *)));
1833         assert(IS_P2ALIGNED(ipv6RouteEntrySize,
1834             sizeof (mib2_ip6RouteEntry_t *)));
1835         assert(IS_P2ALIGNED(ipv6NetToMediaEntrySize,
1836             sizeof (mib2_ip6NetToMediaEntry_t *)));
1837         assert(IS_P2ALIGNED(ipv6MemberEntrySize,
1838             sizeof (ipv6_member_t *)));
1839         assert(IS_P2ALIGNED(ipv6GroupSourceEntrySize,
1840             sizeof (ipv6_grpsrc_t *)));
1841         break;
1842     }
1843     case MIB2_ICMP6: {
1844         mib2_ip6IfIcmpEntry_t *icmp6;
1845         /* Just use the first entry */

```

```

1847         icmp6 = (mib2_ipv6IfIcmpEntry_t *)item->valp;
1848         ipv6IfIcmpEntrySize = icmp6->ipv6IfIcmpEntrySize;
1849         assert(IS_P2ALIGNED(ipv6IfIcmpEntrySize,
1850             sizeof (mib2_ipv6IfIcmpEntry_t *)));
1851         break;
1852     }
1853     case MIB2_TCP: {
1854         mib2_tcp_t      *tcp = (mib2_tcp_t *)item->valp;
1855
1856         tcpConnEntrySize = tcp->tcpConnTableSize;
1857         tcp6ConnEntrySize = tcp->tcp6ConnTableSize;
1858         assert(IS_P2ALIGNED(tcpConnEntrySize,
1859             sizeof (mib2_tcpConnEntry_t *)));
1860         assert(IS_P2ALIGNED(tcp6ConnEntrySize,
1861             sizeof (mib2_tcp6ConnEntry_t *)));
1862         break;
1863     }
1864     case MIB2_UDP: {
1865         mib2_udp_t      *udp = (mib2_udp_t *)item->valp;
1866
1867         udpEntrySize = udp->udpEntrySize;
1868         udp6EntrySize = udp->udp6EntrySize;
1869         assert(IS_P2ALIGNED(udpEntrySize,
1870             sizeof (mib2_udpEntry_t *)));
1871         assert(IS_P2ALIGNED(udp6EntrySize,
1872             sizeof (mib2_udp6Entry_t *)));
1873         break;
1874     }
1875     case MIB2_SCTP: {
1876         mib2_sctp_t      *sctp = (mib2_sctp_t *)item->valp;
1877
1878         sctpEntrySize = sctp->sctpEntrySize;
1879         sctpLocalEntrySize = sctp->sctpLocalEntrySize;
1880         sctpRemoteEntrySize = sctp->sctpRemoteEntrySize;
1881         break;
1882     }
1883     case MIB2_DCCP: {
1884         mib2_dccp_t      *dccp = (mib2_dccp_t *)item->valp;
1885
1886         dccpEntrySize = dccp->dccpConnTableSize;
1887         dccp6EntrySize = dccp->dccp6ConnTableSize;
1888         assert(IS_P2ALIGNED(dccpEntrySize,
1889             sizeof (mib2_dccpConnEntry_t *)));
1890         assert(IS_P2ALIGNED(dccp6EntrySize,
1891             sizeof (mib2_dccp6ConnEntry_t *)));
1892         break;
1893     }
1894 #endif /* ! codereview */
1895 }
1896 } /* 'for' loop 1 ends */
1897
1898 if (Xflag) {
1899     (void) puts("mib_get_constants:");
1900     (void) printf("\tipv6IfStatsEntrySize %d\n",
1901         ipv6IfStatsEntrySize);
1902     (void) printf("\tipAddrEntrySize %d\n", ipAddrEntrySize);
1903     (void) printf("\tipRouteEntrySize %d\n", ipRouteEntrySize);
1904     (void) printf("\tipNetToMediaEntrySize %d\n",
1905         ipNetToMediaEntrySize);
1906     (void) printf("\tipMemberEntrySize %d\n", ipMemberEntrySize);
1907     (void) printf("\tipRouteAttributeSize %d\n",
1908         ipRouteAttributeSize);
1909     (void) printf("\tvifctlSize %d\n", vifctlSize);
1910     (void) printf("\tmfctlSize %d\n", mfctlSize);

```

```

1912         (void) printf("\tipv6AddrEntrySize %d\n", ipv6AddrEntrySize);
1913         (void) printf("\tipv6RouteEntrySize %d\n", ipv6RouteEntrySize);
1914         (void) printf("\tipv6NetToMediaEntrySize %d\n",
1915             ipv6NetToMediaEntrySize);
1916         (void) printf("\tipv6MemberEntrySize %d\n",
1917             ipv6MemberEntrySize);
1918         (void) printf("\tipv6IfIcmpEntrySize %d\n",
1919             ipv6IfIcmpEntrySize);
1920         (void) printf("\tipDestEntrySize %d\n", ipDestEntrySize);
1921         (void) printf("\ttransportMLPSize %d\n", transportMLPSize);
1922         (void) printf("\ttcpConnEntrySize %d\n", tcpConnEntrySize);
1923         (void) printf("\ttcp6ConnEntrySize %d\n", tcp6ConnEntrySize);
1924         (void) printf("\tudpEntrySize %d\n", udpEntrySize);
1925         (void) printf("\tudp6EntrySize %d\n", udp6EntrySize);
1926         (void) printf("\tsctpEntrySize %d\n", sctpEntrySize);
1927         (void) printf("\tsctpLocalEntrySize %d\n", sctpLocalEntrySize);
1928         (void) printf("\tsctpRemoteEntrySize %d\n",
1929             sctpRemoteEntrySize);
1930         (void) printf("\tdccpEntrySize %d\n", dccpEntrySize);
1931         (void) printf("\ttcp6EntrySize %d\n", dccp6EntrySize);
1932 #endif /* ! codereview */
1933     }
1934 }
1935
1936 /* ----- STAT_REPORT ----- */
1937
1938 static void
1939 stat_report(mib_item_t *item)
1940 {
1941     int      jtemp = 0;
1942     char     ifname[LIFNAMSIZ + 1];
1943
1944     /* 'for' loop 1: */
1945     for (; item; item = item->next_item) {
1946         if (Xflag) {
1947             (void) printf("\n--- Entry %d ---\n", ++jtemp);
1948             (void) printf("Group = %d, mib_id = %d, "
1949                 "length = %d, valp = 0x%p\n",
1950                 item->group, item->mib_id,
1951                 item->length, item->valp);
1952         }
1953         if (item->mib_id != 0)
1954             continue; /* 'for' loop 1 */
1955
1956         switch (item->group) {
1957             case MIB2_IP: {
1958                 mib2_ip_t      *ip = (mib2_ip_t *)item->valp;
1959
1960                 if (protocol_selected(IPPROTO_IP) &&
1961                     family_selected(AF_INET)) {
1962                     (void) fputs(v4compat ? "\nIP" : "\nIPv4",
1963                         stdout);
1964                     print_ip_stats(ip);
1965                 }
1966                 break;
1967             }
1968             case MIB2_ICMP: {
1969                 mib2_icmp_t      *icmp =
1970                     (mib2_icmp_t *)item->valp;
1971
1972                 if (protocol_selected(IPPROTO_ICMP) &&
1973                     family_selected(AF_INET)) {
1974                     (void) fputs(v4compat ? "\nICMP" : "\nICMPv4",
1975                         stdout);
1976                     print_icmp_stats(icmp);
1977                 }

```

```

1978         }
1979         break;
1980     }
1981     case MIB2_IP6: {
1982         mib2_ip6IfStatsEntry_t *ip6;
1983         mib2_ip6IfStatsEntry_t sum6;
1984
1985         if (!(protocol_selected(IPPROTO_IPV6)) ||
1986             !(family_selected(AF_INET6)))
1987             break;
1988         bzero(&sum6, sizeof(sum6));
1989         /* 'for' loop 2a: */
1990         for (ip6 = (mib2_ip6IfStatsEntry_t *)item->valp;
1991              (char *)ip6 < (char *)item->valp + item->length;
1992              /* LINTED: (note 1) */)
1993             ip6 = (mib2_ip6IfStatsEntry_t *)((char *)ip6 +
1994                ipv6IfStatsEntrySize) {
1995                 if (ip6->ip6IfIndex == 0) {
1996                     /*
1997                      * The "unknown interface" ip6
1998                      * mib. Just add to the sum.
1999                      */
2000                     sum_ip6_stats(ip6, &sum6);
2001                     continue; /* 'for' loop 2a */
2002                 }
2003                 if (Aflag) {
2004                     (void) printf("\nIPv6 for %s\n",
2005                        ifindex2str(ip6->ip6IfIndex,
2006                        ifname));
2007                     print_ip6_stats(ip6);
2008                 }
2009                 sum_ip6_stats(ip6, &sum6);
2010             } /* 'for' loop 2a ends */
2011             (void) fputs("\nIPv6", stdout);
2012             print_ip6_stats(&sum6);
2013             break;
2014     }
2015     case MIB2_ICMP6: {
2016         mib2_ip6IfIcmpEntry_t *icmp6;
2017         mib2_ip6IfIcmpEntry_t sum6;
2018
2019         if (!(protocol_selected(IPPROTO_ICMPV6)) ||
2020             !(family_selected(AF_INET6)))
2021             break;
2022         bzero(&sum6, sizeof(sum6));
2023         /* 'for' loop 2b: */
2024         for (icmp6 = (mib2_ip6IfIcmpEntry_t *)item->valp;
2025              (char *)icmp6 < (char *)item->valp + item->length;
2026              icmp6 = (void *)((char *)icmp6 +
2027                 ipv6IfIcmpEntrySize)) {
2028             if (icmp6->ip6IfIcmpIfIndex == 0) {
2029                 /*
2030                  * The "unknown interface" icmp6
2031                  * mib. Just add to the sum.
2032                  */
2033                 sum_icmp6_stats(icmp6, &sum6);
2034                 continue; /* 'for' loop 2b: */
2035             }
2036             if (Aflag) {
2037                 (void) printf("\nICMPv6 for %s\n",
2038                    ifindex2str(
2039                    icmp6->ip6IfIcmpIfIndex, ifname));
2040                 print_icmp6_stats(icmp6);
2041             }
2042             sum_icmp6_stats(icmp6, &sum6);
2043         } /* 'for' loop 2b ends */

```

```

2044         (void) fputs("\nICMPv6", stdout);
2045         print_icmp6_stats(&sum6);
2046         break;
2047     }
2048     case MIB2_TCP: {
2049         mib2_tcp_t *tcp = (mib2_tcp_t *)item->valp;
2050
2051         if (protocol_selected(IPPROTO_TCP) &&
2052             (family_selected(AF_INET) ||
2053              family_selected(AF_INET6))) {
2054             (void) fputs("\nTCP", stdout);
2055             print_tcp_stats(tcp);
2056         }
2057         break;
2058     }
2059     case MIB2_UDP: {
2060         mib2_udp_t *udp = (mib2_udp_t *)item->valp;
2061
2062         if (protocol_selected(IPPROTO_UDP) &&
2063             (family_selected(AF_INET) ||
2064              family_selected(AF_INET6))) {
2065             (void) fputs("\nUDP", stdout);
2066             print_udp_stats(udp);
2067         }
2068         break;
2069     }
2070     case MIB2_SCTP: {
2071         mib2_sctp_t *sctp = (mib2_sctp_t *)item->valp;
2072
2073         if (protocol_selected(IPPROTO_SCTP) &&
2074             (family_selected(AF_INET) ||
2075              family_selected(AF_INET6))) {
2076             (void) fputs("\nSCTP", stdout);
2077             print_sctp_stats(sctp);
2078         }
2079         break;
2080     }
2081     case EXPER_RAWIP: {
2082         mib2_rawip_t *rawip =
2083             (mib2_rawip_t *)item->valp;
2084
2085         if (protocol_selected(IPPROTO_RAW) &&
2086             (family_selected(AF_INET) ||
2087              family_selected(AF_INET6))) {
2088             (void) fputs("\nRAWIP", stdout);
2089             print_rawip_stats(rawip);
2090         }
2091         break;
2092     }
2093     case EXPER_IGMP: {
2094         struct igmpstat *igps =
2095             (struct igmpstat *)item->valp;
2096
2097         if (protocol_selected(IPPROTO_IGMP) &&
2098             (family_selected(AF_INET))) {
2099             (void) fputs("\nIGMP:\n", stdout);
2100             print_igmp_stats(igps);
2101         }
2102         break;
2103     }
2104 }
2105 } /* 'for' loop 1 ends */
2106 (void) putchar('\n');
2107 (void) fflush(stdout);
2108 }

```



```

2110 static void
2111 print_ip_stats(mib2_ip_t *ip)
2112 {
2113     prval_init();
2114     pr_int_val("ipForwarding", ip->ipForwarding);
2115     pr_int_val("ipDefaultTTL", ip->ipDefaultTTL);
2116     prval("ipInReceives", ip->ipInReceives);
2117     prval("ipInHdrErrors", ip->ipInHdrErrors);
2118     prval("ipInAddrErrors", ip->ipInAddrErrors);
2119     prval("ipInCksumErrs", ip->ipInCksumErrs);
2120     prval("ipForwDatagrams", ip->ipForwDatagrams);
2121     prval("ipForwProhibits", ip->ipForwProhibits);
2122     prval("ipInUnknownProtos", ip->ipInUnknownProtos);
2123     prval("ipInDiscards", ip->ipInDiscards);
2124     prval("ipInDelivers", ip->ipInDelivers);
2125     prval("ipOutRequests", ip->ipOutRequests);
2126     prval("ipOutDiscards", ip->ipOutDiscards);
2127     prval("ipOutNoRoutes", ip->ipOutNoRoutes);
2128     pr_int_val("ipReasmTimeout", ip->ipReasmTimeout);
2129     prval("ipReasmReqds", ip->ipReasmReqds);
2130     prval("ipReasmOKs", ip->ipReasmOKs);
2131     prval("ipReasmFails", ip->ipReasmFails);
2132     prval("ipReasmDuplicates", ip->ipReasmDuplicates);
2133     prval("ipReasmPartDups", ip->ipReasmPartDups);
2134     prval("ipFragOKs", ip->ipFragOKs);
2135     prval("ipFragFails", ip->ipFragFails);
2136     prval("ipFragCreates", ip->ipFragCreates);
2137     prval("ipRoutingDiscards", ip->ipRoutingDiscards);
2138
2139     prval("tcpInErrs", ip->tcpInErrs);
2140     prval("udpNoPorts", ip->udpNoPorts);
2141     prval("udpInCksumErrs", ip->udpInCksumErrs);
2142     prval("udpInOverflows", ip->udpInOverflows);
2143     prval("rawipInOverflows", ip->rawipInOverflows);
2144     prval("ipsecInSucceeded", ip->ipsecInSucceeded);
2145     prval("ipsecInFailed", ip->ipsecInFailed);
2146     prval("ipInIPv6", ip->ipInIPv6);
2147     prval("ipOutIPv6", ip->ipOutIPv6);
2148     prval("ipOutSwitchIPv6", ip->ipOutSwitchIPv6);
2149     prval_end();
2150 }
2151
2152 static void
2153 print_icmp_stats(mib2_icmp_t *icmp)
2154 {
2155     prval_init();
2156     prval("icmpInMsgs", icmp->icmpInMsgs);
2157     prval("icmpInErrors", icmp->icmpInErrors);
2158     prval("icmpInCksumErrs", icmp->icmpInCksumErrs);
2159     prval("icmpInUnknowns", icmp->icmpInUnknowns);
2160     prval("icmpInDestUnreachs", icmp->icmpInDestUnreachs);
2161     prval("icmpInTimeExcds", icmp->icmpInTimeExcds);
2162     prval("icmpInParmProbs", icmp->icmpInParmProbs);
2163     prval("icmpInSrcQuenchs", icmp->icmpInSrcQuenchs);
2164     prval("icmpInRedirects", icmp->icmpInRedirects);
2165     prval("icmpInBadRedirects", icmp->icmpInBadRedirects);
2166     prval("icmpInEchos", icmp->icmpInEchos);
2167     prval("icmpInEchoReps", icmp->icmpInEchoReps);
2168     prval("icmpInTimestamps", icmp->icmpInTimestamps);
2169     prval("icmpInTimestampReps", icmp->icmpInTimestampReps);
2170     prval("icmpInAddrMasks", icmp->icmpInAddrMasks);
2171     prval("icmpInAddrMaskReps", icmp->icmpInAddrMaskReps);
2172     prval("icmpInFragNeeded", icmp->icmpInFragNeeded);
2173     prval("icmpOutMsgs", icmp->icmpOutMsgs);
2174     prval("icmpOutDrops", icmp->icmpOutDrops);
2175     prval("icmpOutErrors", icmp->icmpOutErrors);

```

```

2176     prval("icmpOutDestUnreachs", icmp->icmpOutDestUnreachs);
2177     prval("icmpOutTimeExcds", icmp->icmpOutTimeExcds);
2178     prval("icmpOutParmProbs", icmp->icmpOutParmProbs);
2179     prval("icmpOutSrcQuenchs", icmp->icmpOutSrcQuenchs);
2180     prval("icmpOutRedirects", icmp->icmpOutRedirects);
2181     prval("icmpOutEchos", icmp->icmpOutEchos);
2182     prval("icmpOutEchoReps", icmp->icmpOutEchoReps);
2183     prval("icmpOutTimestamps", icmp->icmpOutTimestamps);
2184     prval("icmpOutTimestampReps", icmp->icmpOutTimestampReps);
2185     prval("icmpOutAddrMasks", icmp->icmpOutAddrMasks);
2186     prval("icmpOutAddrMaskReps", icmp->icmpOutAddrMaskReps);
2187     prval("icmpOutFragNeeded", icmp->icmpOutFragNeeded);
2188     prval("icmpInOverflows", icmp->icmpInOverflows);
2189     prval_end();
2190 }
2191
2192 static void
2193 print_ip6_stats(mib2_ip6IfStatsEntry_t *ip6)
2194 {
2195     prval_init();
2196     prval("ip6v6Forwarding", ip6->ip6v6Forwarding);
2197     prval("ip6v6DefaultHopLimit", ip6->ip6v6DefaultHopLimit);
2198
2199     prval("ip6v6InReceives", ip6->ip6v6InReceives);
2200     prval("ip6v6InHdrErrors", ip6->ip6v6InHdrErrors);
2201     prval("ip6v6InTooBigErrors", ip6->ip6v6InTooBigErrors);
2202     prval("ip6v6InNoRoutes", ip6->ip6v6InNoRoutes);
2203     prval("ip6v6InAddrErrors", ip6->ip6v6InAddrErrors);
2204     prval("ip6v6InUnknownProtos", ip6->ip6v6InUnknownProtos);
2205     prval("ip6v6InTruncatedPkts", ip6->ip6v6InTruncatedPkts);
2206     prval("ip6v6InDiscards", ip6->ip6v6InDiscards);
2207     prval("ip6v6InDelivers", ip6->ip6v6InDelivers);
2208     prval("ip6v6OutForwDatagrams", ip6->ip6v6OutForwDatagrams);
2209     prval("ip6v6OutRequests", ip6->ip6v6OutRequests);
2210     prval("ip6v6OutDiscards", ip6->ip6v6OutDiscards);
2211     prval("ip6v6OutNoRoutes", ip6->ip6v6OutNoRoutes);
2212     prval("ip6v6OutFragOKs", ip6->ip6v6OutFragOKs);
2213     prval("ip6v6OutFragFails", ip6->ip6v6OutFragFails);
2214     prval("ip6v6OutFragCreates", ip6->ip6v6OutFragCreates);
2215     prval("ip6v6ReasmReqds", ip6->ip6v6ReasmReqds);
2216     prval("ip6v6ReasmOKs", ip6->ip6v6ReasmOKs);
2217     prval("ip6v6ReasmFails", ip6->ip6v6ReasmFails);
2218     prval("ip6v6InMcastPkts", ip6->ip6v6InMcastPkts);
2219     prval("ip6v6OutMcastPkts", ip6->ip6v6OutMcastPkts);
2220     prval("ip6v6ReasmDuplicates", ip6->ip6v6ReasmDuplicates);
2221     prval("ip6v6ReasmPartDups", ip6->ip6v6ReasmPartDups);
2222     prval("ip6v6ForwProhibits", ip6->ip6v6ForwProhibits);
2223     prval("udpInCksumErrs", ip6->udpInCksumErrs);
2224     prval("udpInOverflows", ip6->udpInOverflows);
2225     prval("rawipInOverflows", ip6->rawipInOverflows);
2226     prval("ip6v6InIPv4", ip6->ip6v6InIPv4);
2227     prval("ip6v6OutIPv4", ip6->ip6v6OutIPv4);
2228     prval("ip6v6OutSwitchIPv4", ip6->ip6v6OutSwitchIPv4);
2229     prval_end();
2230 }
2231
2232 static void
2233 print_icmp6_stats(mib2_ip6IfIcmpEntry_t *icmp6)
2234 {
2235     prval_init();
2236     prval("icmp6InMsgs", icmp6->ip6v6IfIcmpInMsgs);
2237     prval("icmp6InErrors", icmp6->ip6v6IfIcmpInErrors);
2238     prval("icmp6InDestUnreachs", icmp6->ip6v6IfIcmpInDestUnreachs);
2239     prval("icmp6InAdminProhibs", icmp6->ip6v6IfIcmpInAdminProhibs);
2240     prval("icmp6InTimeExcds", icmp6->ip6v6IfIcmpInTimeExcds);
2241     prval("icmp6InParmProblems", icmp6->ip6v6IfIcmpInParmProblems);

```

```

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

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

```

```

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

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

```

```

2374     prval("tcpInClosed",          tcp->tcpInClosed);
2375     prval("tcpRttNoUpdate",        tcp->tcpRttNoUpdate);
2376     prval("tcpRttUpdate",          tcp->tcpRttUpdate);
2377     prval("tcpTimRetrans",         tcp->tcpTimRetrans);
2378     prval("tcpTimRetransDrop",     tcp->tcpTimRetransDrop);
2379     prval("tcpTimKeepalive",       tcp->tcpTimKeepalive);
2380     prval("tcpTimKeepaliveProbe",  tcp->tcpTimKeepaliveProbe);
2381     prval("tcpTimKeepaliveDrop",   tcp->tcpTimKeepaliveDrop);
2382     prval("tcpListenDrop",         tcp->tcpListenDrop);
2383     prval("tcpListenDropQ0",       tcp->tcpListenDropQ0);
2384     prval("tcpHalfOpenDrop",       tcp->tcpHalfOpenDrop);
2385     prval("tcpOutSackRetrans",     tcp->tcpOutSackRetransSegs);
2386     prval_end();
2388 }

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

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

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

2439 static void

```

```

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

2489 static void
2490 sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6, mib2_ipv6IfStatsEntry_t *sum6)
2491 {
2492     /* First few are not additive */
2493     sum6->ip6Forwarding = ip6->ip6Forwarding;
2494     sum6->ip6DefaultHopLimit = ip6->ip6DefaultHopLimit;

2496     sum6->ip6InReceives += ip6->ip6InReceives;
2497     sum6->ip6InHdrErrors += ip6->ip6InHdrErrors;
2498     sum6->ip6InTooBigErrors += ip6->ip6InTooBigErrors;
2499     sum6->ip6InNoRoutes += ip6->ip6InNoRoutes;
2500     sum6->ip6InAddrErrors += ip6->ip6InAddrErrors;
2501     sum6->ip6InUnknownProtos += ip6->ip6InUnknownProtos;
2502     sum6->ip6InTruncatedPkts += ip6->ip6InTruncatedPkts;
2503     sum6->ip6InDiscards += ip6->ip6InDiscards;
2504     sum6->ip6InDelivers += ip6->ip6InDelivers;
2505     sum6->ip6OutForwDatagrams += ip6->ip6OutForwDatagrams;

```

```

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

2525 static void
2526 sum_icmp6_stats(mib2_ip6IfIcmpEntry_t *icmp6, mib2_ip6IfIcmpEntry_t *sum6)
2527 {
2528     sum6->ipv6IfIcmpInMsgs += icmp6->ipv6IfIcmpInMsgs;
2529     sum6->ipv6IfIcmpInErrors += icmp6->ipv6IfIcmpInErrors;
2530     sum6->ipv6IfIcmpInDestUnreaches += icmp6->ipv6IfIcmpInDestUnreaches;
2531     sum6->ipv6IfIcmpInAdminProhibs += icmp6->ipv6IfIcmpInAdminProhibs;
2532     sum6->ipv6IfIcmpInTimeExcds += icmp6->ipv6IfIcmpInTimeExcds;
2533     sum6->ipv6IfIcmpInParmProblems += icmp6->ipv6IfIcmpInParmProblems;
2534     sum6->ipv6IfIcmpInPktTooBigs += icmp6->ipv6IfIcmpInPktTooBigs;
2535     sum6->ipv6IfIcmpInEchos += icmp6->ipv6IfIcmpInEchos;
2536     sum6->ipv6IfIcmpInEchoReplies += icmp6->ipv6IfIcmpInEchoReplies;
2537     sum6->ipv6IfIcmpInRouterSolicits += icmp6->ipv6IfIcmpInRouterSolicits;
2538     sum6->ipv6IfIcmpInRouterAdvertisements +=
2539         icmp6->ipv6IfIcmpInRouterAdvertisements;
2540     sum6->ipv6IfIcmpInNeighborSolicits +=
2541         icmp6->ipv6IfIcmpInNeighborSolicits;
2542     sum6->ipv6IfIcmpInNeighborAdvertisements +=
2543         icmp6->ipv6IfIcmpInNeighborAdvertisements;
2544     sum6->ipv6IfIcmpInRedirects += icmp6->ipv6IfIcmpInRedirects;
2545     sum6->ipv6IfIcmpInGroupMembQueries +=
2546         icmp6->ipv6IfIcmpInGroupMembQueries;
2547     sum6->ipv6IfIcmpInGroupMembResponses +=
2548         icmp6->ipv6IfIcmpInGroupMembResponses;
2549     sum6->ipv6IfIcmpInGroupMembReductions +=
2550         icmp6->ipv6IfIcmpInGroupMembReductions;
2551     sum6->ipv6IfIcmpOutMsgs += icmp6->ipv6IfIcmpOutMsgs;
2552     sum6->ipv6IfIcmpOutErrors += icmp6->ipv6IfIcmpOutErrors;
2553     sum6->ipv6IfIcmpOutDestUnreaches += icmp6->ipv6IfIcmpOutDestUnreaches;
2554     sum6->ipv6IfIcmpOutAdminProhibs += icmp6->ipv6IfIcmpOutAdminProhibs;
2555     sum6->ipv6IfIcmpOutTimeExcds += icmp6->ipv6IfIcmpOutTimeExcds;
2556     sum6->ipv6IfIcmpOutParmProblems += icmp6->ipv6IfIcmpOutParmProblems;
2557     sum6->ipv6IfIcmpOutPktTooBigs += icmp6->ipv6IfIcmpOutPktTooBigs;
2558     sum6->ipv6IfIcmpOutEchos += icmp6->ipv6IfIcmpOutEchos;
2559     sum6->ipv6IfIcmpOutEchoReplies += icmp6->ipv6IfIcmpOutEchoReplies;
2560     sum6->ipv6IfIcmpOutRouterSolicits +=
2561         icmp6->ipv6IfIcmpOutRouterSolicits;
2562     sum6->ipv6IfIcmpOutRouterAdvertisements +=
2563         icmp6->ipv6IfIcmpOutRouterAdvertisements;
2564     sum6->ipv6IfIcmpOutNeighborSolicits +=
2565         icmp6->ipv6IfIcmpOutNeighborSolicits;
2566     sum6->ipv6IfIcmpOutNeighborAdvertisements +=
2567         icmp6->ipv6IfIcmpOutNeighborAdvertisements;
2568     sum6->ipv6IfIcmpOutRedirects += icmp6->ipv6IfIcmpOutRedirects;
2569     sum6->ipv6IfIcmpOutGroupMembQueries +=
2570         icmp6->ipv6IfIcmpOutGroupMembQueries;
2571     sum6->ipv6IfIcmpOutGroupMembResponses +=

```

```

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

```

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

2747                 first = B_FALSE;
2748             }
2749             if_report_ip4(ap, ifname,
2750                 logintname, &stat, B_TRUE);
2751             ifindex_v4 = new_ifindex;
2752         } else {
2753             if_report_ip4(ap, ifname,
2754                 logintname, &stat, B_FALSE);
2755         }
2756     } /* 'for' loop 2a ends */
2757 } else if (!alreadydone) {
2758     char ifname[LIFNAMSIZ + 1];
2759     char buf[LIFNAMSIZ + 1];
2760     mib2_ipAddrEntry_t *ap;
2761     struct ifstat t;
2762     struct iflist *t1p = NULL;
2763     struct iflist *nextnew = &newlist;
2764     struct iflist *walkold;
2765     struct iflist *cleanlist;
2766     boolean_t found_if = B_FALSE;

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

```

```

2770      /*
2771       * Check if there is anything to do.
2772       */
2773      if (item->length <
2774          sizeof (mib2_ipAddrEntry_t)) {
2775          fail(0, "No compatible interfaces");
2776      }
2777
2778      /*
2779       * 'for' loop 2b: find the "right" entry:
2780       * If an interface name to match has been
2781       * supplied then try and find it, otherwise
2782       * match the first non-loopback interface found.
2783       * Use lo0 if all else fails.
2784       */
2785      for (ap = (mib2_ipAddrEntry_t *)item->valp;
2786           (char *)ap < (char *)item->valp
2787             + item->length;
2788           ap++) {
2789          (void) octetstr(&ap->ipAdEntIfIndex,
2790                        'a', ifname, sizeof (ifname));
2791          (void) strtok(ifname, ":");
2792
2793          if (matchname) {
2794              if (strcmp(matchname,
2795                          ifname) == 0) {
2796                  /* 'for' loop 2b */
2797                  found_if = B_TRUE;
2798                  break;
2799              }
2800              } else if (strcmp(ifname, "lo0") != 0)
2801                  break; /* 'for' loop 2b */
2802      } /* 'for' loop 2b ends */
2803
2804      if (matchname == NULL) {
2805          matchname = ifname;
2806      } else {
2807          if (!found_if)
2808              fail(0, "-I: %s no such "
2809                  "interface.", matchname);
2810      }
2811
2812      if (Iflag_only == 0 || !reentry) {
2813          (void) printf("    input    %-6.6s    "
2814                        "output    ",
2815                        matchname);
2816          (void) printf("    input    (Total)    "
2817                        "output\n");
2818          (void) printf("%-7.7s %-5.5s %-7.7s "
2819                        "%-5.5s %-6.6s ",
2820                        "packets", "errs", "packets",
2821                        "errs", "colls");
2822          (void) printf("%-7.7s %-5.5s %-7.7s "
2823                        "%-5.5s %-6.6s\n",
2824                        "packets", "errs", "packets",
2825                        "errs", "colls");
2826      }
2827
2828      sum = zerostat;
2829
2830      /* 'for' loop 2c: */
2831      for (ap = (mib2_ipAddrEntry_t *)item->valp;
2832           (char *)ap < (char *)item->valp
2833             + item->length;
2834           ap++) {
2835          (void) octetstr(&ap->ipAdEntIfIndex,

```

```

2836      'a', buf, sizeof (buf));
2837      (void) strtok(buf, ":");
2838
2839      /*
2840       * We have reduced the IP interface
2841       * name, which could have been a
2842       * logical, down to a name suitable
2843       * for use with kstats.
2844       * We treat this name as unique and
2845       * only collate statistics for it once
2846       * per pass. This is to avoid falsely
2847       * amplifying these statistics by the
2848       * the number of logical instances.
2849       */
2850      if ((t1p != NULL) &&
2851          ((strcmp(buf, t1p->ifname) == 0)) {
2852          continue;
2853      }
2854
2855      /*
2856       * First lookup the "link" kstats in
2857       * case the link is renamed. Then
2858       * fallback to the legacy kstats for
2859       * those non-GLDv3 links.
2860       */
2861      if (((ksp = kstat_lookup(kc, "link",
2862                              0, buf)) != NULL ||
2863          (ksp = kstat_lookup(kc, NULL, -1,
2864                              buf)) != NULL) && (ksp->ks_type ==
2865          KSTAT_TYPE_NAMED)) {
2866          (void) safe_kstat_read(kc, ksp,
2867                                NULL);
2868      }
2869
2870      t.ipackets = kstat_named_value(ksp,
2871                                     "ipackets");
2872      t.ierrors = kstat_named_value(ksp,
2873                                     "ierrors");
2874      t.opackets = kstat_named_value(ksp,
2875                                     "opackets");
2876      t.oerrors = kstat_named_value(ksp,
2877                                     "oerrors");
2878      t.collisions = kstat_named_value(ksp,
2879                                       "collisions");
2880
2881      if (strcmp(buf, matchname) == 0)
2882          new = t;
2883
2884      /* Build the interface list */
2885
2886      t1p = malloc(sizeof (struct iflist));
2887      (void) strcpy(t1p->ifname, buf,
2888                  sizeof (t1p->ifname));
2889      t1p->tot = t;
2890      *nextnew = t1p;
2891      nextnew = &t1p->next_if;
2892
2893      /*
2894       * First time through.
2895       * Just add up the interface stats.
2896       */
2897
2898      if (oldlist == NULL) {
2899          if_stat_total(&zerostat,
2900                      &t, &sum);
2901          continue;

```

```

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

```

```

2968         while (cleanlist != NULL) {
2969             tlp = cleanlist->next_if;
2970             free(cleanlist);
2971             cleanlist = tlp;
2972         }
2973     }
2974     break;
2975
2976     }
2977     case MIB2_IP6:
2978     if (item->mib_id != MIB2_IP6_ADDR ||
2979         !family_selected(AF_INET6))
2980         continue; /* 'for' loop 1 */
2981
2982     {
2983         static struct ifstat old6 = {0L, 0L, 0L, 0L, 0L};
2984         static struct ifstat new6 = {0L, 0L, 0L, 0L, 0L};
2985         struct ifstat sum6;
2986         struct iflist *newlist6 = NULL;
2987         static struct iflist *oldlist6 = NULL;
2988         kstat_t *ksp;
2989
2990         if (once_only) {
2991             char ifname[LIFNAMSIZ + 1];
2992             char logintname[LIFNAMSIZ + 1];
2993             mib2_ipv6AddrEntry_t *ap6;
2994             struct ifstat stat = {0L, 0L, 0L, 0L, 0L};
2995             boolean_t first = B_TRUE;
2996             uint32_t new_ifindex;
2997
2998             if (Xflag)
2999                 (void) printf("if_report: %d items\n",
3000                               (item->length)
3001                               / sizeof (mib2_ipv6AddrEntry_t));
3002             /* 'for' loop 2d: */
3003             for (ap6 = (mib2_ipv6AddrEntry_t *)item->valp;
3004                  (char *)ap6 < (char *)item->valp
3005                  + item->length;
3006                  ap6++) {
3007                 (void) octetstr(&ap6->ipv6AddrIfIndex,
3008                                'a', logintname,
3009                                sizeof (logintname));
3010                 (void) strcpy(ifname, logintname);
3011                 (void) strtok(ifname, ":");
3012                 if (matchname != NULL &&
3013                     strcmp(matchname, ifname) != 0 &&
3014                     strcmp(matchname, logintname) != 0)
3015                     continue; /* 'for' loop 2d */
3016                 new_ifindex =
3017                     if_nameindex(logintname);
3018
3019                 /*
3020                  * First lookup the "link" kstats in
3021                  * case the link is renamed. Then
3022                  * fallback to the legacy kstats for
3023                  * those non-GLDv3 links.
3024                  */
3025                 if (new_ifindex != ifindex_v6 &&
3026                     ((ksp = kstat_lookup(kc, "link", 0,
3027                                           ifname)) != NULL ||
3028                      (ksp = kstat_lookup(kc, NULL, -1,
3029                                           ifname)) != NULL)) {
3029                     (void) safe_kstat_read(kc, ksp,
3030                                             NULL);
3031                     stat.ipackets =
3032                         kstat_named_value(ksp,
3033                                             "ipackets");
3034                     stat.ierrors =

```

```

3034         kstat_named_value(ksp,
3035         "ierrors");
3036     stat.opackets =
3037         kstat_named_value(ksp,
3038         "opackets");
3039     stat.oerrors =
3040         kstat_named_value(ksp,
3041         "oerrors");
3042     stat.collisions =
3043         kstat_named_value(ksp,
3044         "collisions");
3045     if (first) {
3046         if (!first_header)
3047             (void) putchar('\n');
3048         first_header = B_FALSE;
3049         (void) printf(
3050             "%-5.5s %-5.5s%"
3051             "-27.27s %-27.27s "
3052             "%-6.6s %-5.5s "
3053             "%-6.6s %-5.5s "
3054             "%-6.6s\n",
3055             "Name", "Mtu",
3056             "Net/Dest",
3057             "Address", "Ipkts",
3058             "Ierrs", "Opkts",
3059             "Oerrs", "Collis");
3060         first = B_FALSE;
3061     }
3062     if_report_ip6(ap6, ifname,
3063     logintname, &stat, B_TRUE);
3064     ifindex_v6 = new_ifindex;
3065 } else {
3066     if_report_ip6(ap6, ifname,
3067     logintname, &stat, B_FALSE);
3068 }
3069 } /* 'for' loop 2d ends */
3070 } else if (!alreadydone) {
3071     char ifname[LIFNAMSIZ + 1];
3072     char buf[IFNAMSIZ + 1];
3073     mib2_ip6AddrEntry_t *ap6;
3074     struct ifstat t;
3075     struct iflist *t1p = NULL;
3076     struct iflist **nextnew = &newlist6;
3077     struct iflist *walkold;
3078     struct iflist *cleanlist;
3079     boolean_t found_if = B_FALSE;
3081
3082     alreadydone = B_TRUE; /* ignore other case */
3083
3084     /*
3085      * Check if there is anything to do.
3086      */
3087     if (item->length <
3088         sizeof (mib2_ip6AddrEntry_t)) {
3089         fail(0, "No compatible interfaces");
3090     }
3091
3092     /*
3093      * 'for' loop 2e: find the "right" entry:
3094      * If an interface name to match has been
3095      * supplied then try and find it, otherwise
3096      * match the first non-loopback interface found.
3097      * Use lo0 if all else fails.
3098      */
3099     for (ap6 = (mib2_ip6AddrEntry_t *)item->valp;
3100         (char *)ap6 < (char *)item->valp

```

```

3100         + item->length;
3101     ap6++) {
3102         (void) octetstr(&ap6->ip6AddrIfIndex,
3103         'a', ifname, sizeof (ifname));
3104         (void) strtok(ifname, ":");
3106
3107         if (matchname) {
3108             if (strcmp(matchname,
3109             ifname) == 0) {
3110                 /* 'for' loop 2e */
3111                 found_if = B_TRUE;
3112                 break;
3113             } else if (strcmp(ifname, "lo0") != 0)
3114                 break; /* 'for' loop 2e */
3115         } /* 'for' loop 2e ends */
3117
3118         if (matchname == NULL) {
3119             matchname = ifname;
3120         } else {
3121             if (!found_if)
3122                 fail(0, "-I: %s no such "
3123                 "interface.", matchname);
3124         }
3125
3126         if (Iflag_only == 0 || !reentry) {
3127             (void) printf(
3128                 "    input    %-6.6s"
3129                 "    output    ",
3130                 matchname);
3131             (void) printf("    input    (Total)"
3132             "    output\n");
3133             (void) printf("%-7.7s %-5.5s %-7.7s "
3134             "%-5.5s %-6.6s ",
3135             "packets", "errs", "packets",
3136             "errs", "colls");
3137             (void) printf("%-7.7s %-5.5s %-7.7s "
3138             "%-5.5s %-6.6s\n",
3139             "packets", "errs", "packets",
3140             "errs", "colls");
3141         }
3142
3143         sum6 = zerostat;
3144
3145         /* 'for' loop 2f: */
3146         for (ap6 = (mib2_ip6AddrEntry_t *)item->valp;
3147             (char *)ap6 < (char *)item->valp
3148             + item->length;
3149             ap6++) {
3150             (void) octetstr(&ap6->ip6AddrIfIndex,
3151             'a', buf, sizeof (buf));
3152             (void) strtok(buf, ":");
3153
3154             /*
3155              * We have reduced the IP interface
3156              * name, which could have been a
3157              * logical, down to a name suitable
3158              * for use with kstats.
3159              * We treat this name as unique and
3160              * only collate statistics for it once
3161              * per pass. This is to avoid falsely
3162              * amplifying these statistics by the
3163              * the number of logical instances.
3164              */
3165             if ((t1p != NULL) &&

```



```

3166         ((strcmp(buf, tlp->ifname) == 0))) {
3167             continue;
3168         }
3169
3170     /*
3171     * First lookup the "link" kstats in
3172     * case the link is renamed. Then
3173     * fallback to the legacy kstats for
3174     * those non-GLDv3 links.
3175     */
3176     if (((ksp = kstat_lookup(kc, "link",
3177         0, buf)) != NULL ||
3178         (ksp = kstat_lookup(kc, NULL, -1,
3179             buf)) != NULL) && (ksp->ks_type ==
3180             KSTAT_TYPE_NAMED)) {
3181         (void) safe_kstat_read(kc,
3182             ksp, NULL);
3183     }
3184
3185     t.ipackets = kstat_named_value(ksp,
3186         "ipackets");
3187     t.ierrors = kstat_named_value(ksp,
3188         "ierrors");
3189     t.opackets = kstat_named_value(ksp,
3190         "opackets");
3191     t.oerrors = kstat_named_value(ksp,
3192         "oerrors");
3193     t.collisions = kstat_named_value(ksp,
3194         "collisions");
3195
3196     if (strcmp(buf, matchname) == 0)
3197         new6 = t;
3198
3199     /* Build the interface list */
3200
3201     tlp = malloc(sizeof (struct iflist));
3202     (void) strcpy(tlp->ifname, buf,
3203         sizeof (tlp->ifname));
3204     tlp->tot = t;
3205     *nextnew = tlp;
3206     nextnew = &tlp->next_if;
3207
3208     /*
3209     * First time through.
3210     * Just add up the interface stats.
3211     */
3212
3213     if (oldlist6 == NULL) {
3214         if_stat_total(&zerostat,
3215             &t, &sum6);
3216         continue;
3217     }
3218
3219     /*
3220     * Walk old list for the interface.
3221     *
3222     * If found, add difference to total.
3223     *
3224     * If not, an interface has been plumbed
3225     * up. In this case, we will simply
3226     * ignore the new interface until the
3227     * next interval; as there's no easy way
3228     * to acquire statistics between time
3229     * of the plumb and the next interval
3230     * boundary. This results in inaccurate
3231     * total values for current interval.

```

```

3232     *
3233     * Note the case when an interface is
3234     * unplumbed; as similar problems exist.
3235     * The unplumbed interface is not in the
3236     * current list, and there's no easy way
3237     * to account for the statistics between
3238     * the previous interval and time of the
3239     * unplumb. Therefore, we (in a sense)
3240     * ignore the removed interface by only
3241     * involving "current" interfaces when
3242     * computing the total statistics.
3243     * Unfortunately, this also results in
3244     * inaccurate values for interval total.
3245     */
3246
3247     for (walkold = oldlist6;
3248         walkold != NULL;
3249         walkold = walkold->next_if) {
3250         if (strcmp(walkold->ifname,
3251             buf) == 0) {
3252             if_stat_total(
3253                 &walkold->tot,
3254                 &t, &sum6);
3255             break;
3256         }
3257     }
3258
3259     /* 'for' loop 2f ends */
3260
3261     *nextnew = NULL;
3262
3263     (void) printf("%-7llu %-5llu %-7llu "
3264         "%-5llu %-6llu ",
3265         new6.ipackets - old6.ipackets,
3266         new6.ierrors - old6.ierrors,
3267         new6.opackets - old6.opackets,
3268         new6.oerrors - old6.oerrors,
3269         new6.collisions - old6.collisions);
3270
3271     (void) printf("%-7llu %-5llu %-7llu "
3272         "%-5llu %-6llu\n", sum6.ipackets,
3273         sum6.ierrors, sum6.opackets,
3274         sum6.oerrors, sum6.collisions);
3275
3276     /*
3277     * Tidy things up once finished.
3278     */
3279
3280     old6 = new6;
3281     cleanlist = oldlist6;
3282     oldlist6 = newlist6;
3283     while (cleanlist != NULL) {
3284         tlp = cleanlist->next_if;
3285         free(cleanlist);
3286         cleanlist = tlp;
3287     }
3288     break;
3289 }
3290 }
3291 }
3292 (void) fflush(stdout);
3293 } /* 'for' loop 1 ends */
3294 if ((iflag_only == 0) && (!once_only))
3295     (void) putchar('\n');
3296 reentry = B_TRUE;
3297 }

```

```

3299 static void
3300 if_report_ip4(mib2_ipAddrEntry_t *ap,
3301             char ifname[], char loginname[], struct ifstat *statptr,
3302             boolean_t ksp_not_null) {
3303
3304     char abuf[MAXHOSTNAMELEN + 1];
3305     char dstbuf[MAXHOSTNAMELEN + 1];
3306
3307     if (ksp_not_null) {
3308         (void) printf("%-5s %-4u ",
3309                     ifname, ap->ipAdEntInfo.ae_mtu);
3310         if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3311             (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr,
3312                           abuf, sizeof (abuf));
3313         else
3314             (void) pr_netaddr(ap->ipAdEntAddr,
3315                               ap->ipAdEntNetMask, abuf, sizeof (abuf));
3316         (void) printf("%-13s %-14s %-6llu %-5llu %-6llu %-5llu "
3317                     "%-6llu %-6llu\n",
3318                     abuf, pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3319                     statptr->ipackets, statptr->ierrors,
3320                     statptr->opackets, statptr->oerrors,
3321                     statptr->collisions, 0LL);
3322     }
3323     /*
3324      * Print logical interface info if Aflag set (including logical unit 0)
3325      */
3326     if (Aflag) {
3327         *statptr = zerostat;
3328         statptr->ipackets = ap->ipAdEntInfo.ae_ibcnt;
3329         statptr->opackets = ap->ipAdEntInfo.ae_obcnt;
3330
3331         (void) printf("%-5s %-4u ", loginname, ap->ipAdEntInfo.ae_mtu);
3332         if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3333             (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr, abuf,
3334                           sizeof (abuf));
3335         else
3336             (void) pr_netaddr(ap->ipAdEntAddr, ap->ipAdEntNetMask,
3337                               abuf, sizeof (abuf));
3338
3339         (void) printf("%-13s %-14s %-6llu %-5s %-6s "
3340                     "%-5s %-6s %-6llu\n", abuf,
3341                     pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3342                     statptr->ipackets, "N/A", "N/A", "N/A", "N/A",
3343                     0LL);
3344     }
3345 }
3346
3347 static void
3348 if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
3349             char ifname[], char loginname[], struct ifstat *statptr,
3350             boolean_t ksp_not_null) {
3351
3352     char abuf[MAXHOSTNAMELEN + 1];
3353     char dstbuf[MAXHOSTNAMELEN + 1];
3354
3355     if (ksp_not_null) {
3356         (void) printf("%-5s %-4u ", ifname, ap6->ipv6AddrInfo.ae_mtu);
3357         if (ap6->ipv6AddrInfo.ae_flags &
3358             IFF_POINTOPOINT) {
3359             (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3360                             abuf, sizeof (abuf));
3361         } else {
3362             (void) pr_prefix6(&ap6->ipv6AddrAddress,
3363                               ap6->ipv6AddrPfxLength, abuf,

```

```

3364             sizeof (abuf));
3365         }
3366         (void) printf("%-27s %-27s %-6llu %-5llu "
3367                     "%-6llu %-5llu %-6llu\n",
3368                     abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3369                                     sizeof (dstbuf)),
3370                     statptr->ipackets, statptr->ierrors, statptr->opackets,
3371                     statptr->oerrors, statptr->collisions);
3372     }
3373     /*
3374      * Print logical interface info if Aflag set (including logical unit 0)
3375      */
3376     if (Aflag) {
3377         *statptr = zerostat;
3378         statptr->ipackets = ap6->ipv6AddrInfo.ae_ibcnt;
3379         statptr->opackets = ap6->ipv6AddrInfo.ae_obcnt;
3380
3381         (void) printf("%-5s %-4u ", loginname,
3382                     ap6->ipv6AddrInfo.ae_mtu);
3383         if (ap6->ipv6AddrInfo.ae_flags & IFF_POINTOPOINT)
3384             (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3385                             abuf, sizeof (abuf));
3386         else
3387             (void) pr_prefix6(&ap6->ipv6AddrAddress,
3388                               ap6->ipv6AddrPfxLength, abuf, sizeof (abuf));
3389         (void) printf("%-27s %-27s %-6llu %-5s %-6s %-5s %-6s\n",
3390                     abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3391                                     sizeof (dstbuf)),
3392                     statptr->ipackets, "N/A", "N/A", "N/A", "N/A");
3393     }
3394 }
3395
3396 /* ----- DHCP_REPORT (netstat -D) ----- */
3397
3398 static boolean_t
3399 dhcp_do_ipc(dhcp_ipc_type_t type, const char *ifname, boolean_t printed_one)
3400 {
3401     dhcp_ipc_request_t *request;
3402     dhcp_ipc_reply_t *reply;
3403     int error;
3404
3405     request = dhcp_ipc_alloc_request(type, ifname, NULL, 0, DHCP_TYPE_NONE);
3406     if (request == NULL)
3407         fail(0, "dhcp_do_ipc: out of memory");
3408
3409     error = dhcp_ipc_make_request(request, &reply, DHCP_IPC_WAIT_DEFAULT);
3410     if (error != 0) {
3411         free(request);
3412         fail(0, "dhcp_do_ipc: %s", dhcp_ipc_strerror(error));
3413     }
3414
3415     free(request);
3416     error = reply->return_code;
3417     if (error == DHCP_IPC_E_UNKIF) {
3418         free(reply);
3419         return (printed_one);
3420     }
3421     if (error != 0) {
3422         free(reply);
3423         fail(0, "dhcp_do_ipc: %s", dhcp_ipc_strerror(error));
3424     }
3425
3426     if (timestamp_fmt != NODATE)
3427         print_timestamp(timestamp_fmt);
3428
3429     if (!printed_one)

```

```

3430         (void) printf("%s", dhcp_status_hdr_string());
3432         (void) printf("%s", dhcp_status_reply_to_string(reply));
3433         free(reply);
3434         return (B_TRUE);
3435     }
3437     /*
3438     * dhcp_walk_interfaces: walk the list of interfaces for a given address
3439     * family (af).  For each, print out the DHCP status using dhcp_do_ipc.
3440     */
3441     static boolean_t
3442     dhcp_walk_interfaces(int af, boolean_t printed_one)
3443     {
3444         struct lifnum   lifn;
3445         struct lifconf   lifc;
3446         int             n_ifs, i, sock_fd;
3448         sock_fd = socket(af, SOCK_DGRAM, 0);
3449         if (sock_fd == -1)
3450             return (printed_one);
3452         /*
3453         * SIOCGIFNUM is just an estimate.  If the ioctl fails, we don't care;
3454         * just drive on and use SIOCGIFCONF with increasing buffer sizes, as
3455         * is traditional.
3456         */
3457         (void) memset(&lifn, 0, sizeof (lifn));
3458         lifn.lifn_family = af;
3459         lifn.lifn_flags = LIFC_ALLZONES | LIFC_NOXMIT | LIFC_UNDER_IPMP;
3460         if (ioctl(sock_fd, SIOCGIFNUM, &lifn) == -1)
3461             n_ifs = LIFN_GUARD_VALUE;
3462         else
3463             n_ifs = lifn.lifn_count + LIFN_GUARD_VALUE;
3465         (void) memset(&lifc, 0, sizeof (lifc));
3466         lifc.lifc_family = af;
3467         lifc.lifc_flags = lifn.lifn_flags;
3468         lifc.lifc_len = n_ifs * sizeof (struct lifreq);
3469         lifc.lifc_buf = malloc(lifc.lifc_len);
3470         if (lifc.lifc_buf != NULL) {
3472             if (ioctl(sock_fd, SIOCGIFCONF, &lifc) == -1) {
3473                 (void) close(sock_fd);
3474                 free(lifc.lifc_buf);
3475                 return (NULL);
3476             }
3478             n_ifs = lifc.lifc_len / sizeof (struct lifreq);
3480             for (i = 0; i < n_ifs; i++) {
3481                 printed_one = dhcp_do_ipc(DHCP_STATUS |
3482                     (af == AF_INET6 ? DHCP_V6 : 0),
3483                     lifc.lifc_req[i].lifc_name, printed_one);
3484             }
3485             (void) close(sock_fd);
3486             free(lifc.lifc_buf);
3487             return (printed_one);
3488         }
3489     }
3491     static void
3492     dhcp_report(char *ifname)
3493     {
3494         boolean_t printed_one;

```

```

3496         if (!family_selected(AF_INET) && !family_selected(AF_INET6))
3497             return;
3499         printed_one = B_FALSE;
3500         if (ifname != NULL) {
3501             if (family_selected(AF_INET)) {
3502                 printed_one = dhcp_do_ipc(DHCP_STATUS, ifname,
3503                     printed_one);
3504             }
3505             if (family_selected(AF_INET6)) {
3506                 printed_one = dhcp_do_ipc(DHCP_STATUS | DHCP_V6,
3507                     ifname, printed_one);
3508             }
3509             if (!printed_one) {
3510                 fail(0, "%s: %s", ifname,
3511                     dhcp_ipc_strerror(DHCP_IPC_E_UNKIF));
3512             }
3513         } else {
3514             if (family_selected(AF_INET)) {
3515                 printed_one = dhcp_walk_interfaces(AF_INET,
3516                     printed_one);
3517             }
3518             if (family_selected(AF_INET6))
3519                 (void) dhcp_walk_interfaces(AF_INET6, printed_one);
3520         }
3521     }
3523     /* ----- GROUP_REPORT (netstat -g) ----- */
3525     static void
3526     group_report(mib_item_t *item)
3527     {
3528         mib_item_t      *v4grp = NULL, *v4src = NULL;
3529         mib_item_t      *v6grp = NULL, *v6src = NULL;
3530         int             jtemp = 0;
3531         char             ifname[LIFNAMSIZ + 1];
3532         char             abuf[MAXHOSTNAMELEN + 1];
3533         ip_member_t      *ipmp;
3534         ip_grpsrc_t      *ips;
3535         ipv6_member_t    *ipmp6;
3536         ipv6_grpsrc_t    *ips6;
3537         boolean_t        first, first_src;
3539         /* 'for' loop 1: */
3540         for (; item; item = item->next_item) {
3541             if (Xflag) {
3542                 (void) printf("\n--- Entry %d ---\n", ++jtemp);
3543                 (void) printf("Group = %d, mib_id = %d, "
3544                     "length = %d, valp = 0x%p\n",
3545                     item->group, item->mib_id, item->length,
3546                     item->valp);
3547             }
3548             if (item->group == MIB2_IP && family_selected(AF_INET)) {
3549                 switch (item->mib_id) {
3550                     case EXPER_IP_GROUP_MEMBERSHIP:
3551                         v4grp = item;
3552                         if (Xflag)
3553                             (void) printf("item is v4grp info\n");
3554                         break;
3555                     case EXPER_IP_GROUP_SOURCES:
3556                         v4src = item;
3557                         if (Xflag)
3558                             (void) printf("item is v4src info\n");
3559                         break;
3560                     default:
3561                         continue;

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```



```

3826     (char *)dce < (char *)v6dce->valp + v6dce->length;
3827     /* LINTED: (note 1) */
3828     dce = (dest_cache_entry_t *)((char *)dce +
3829     ipDestEntrySize)) {
3830         if (first) {
3831             (void) putchar('\n');
3832             (void) puts("Destination Cache Entries: IPv6");
3833             (void) puts(
3834                 "Address                      PMTU  "
3835                 " Age Flags If ");
3836             (void) puts(
3837                 "-----"
3838                 "-----");
3839             first = B_FALSE;
3840         }
3841
3842         (void) printf("%-27s %6u %5u %-5s %s\n",
3843             pr_addr6(&dce->DestIpv6Address, abuf,
3844             sizeof (abuf)),
3845             dce->DestPmtu, dce->DestAge,
3846             dceflags2str(dce->DestFlags, flbuf),
3847             dce->DestIfindex == 0 ? "" :
3848             ifindex2str(dce->DestIfindex, ifname));
3849     }
3850     (void) fflush(stdout);
3851 }
3852
3854 /* ----- ARP_REPORT (netstat -p) ----- */
3855
3856 static void
3857 arp_report(mib_item_t *item)
3858 {
3859     int             jtemp = 0;
3860     char             ifname[LIFNAMSIZ + 1];
3861     char             abuf[MAXHOSTNAMELEN + 1];
3862     char             maskbuf[STR_EXPAND * OCTET_LENGTH + 1];
3863     char             flbuf[32]; /* ACE_F_flags */
3864     char             xbuf[STR_EXPAND * OCTET_LENGTH + 1];
3865     mib2_ipNetToMediaEntry_t *np;
3866     int             flags;
3867     boolean_t       first;
3868
3869     if (!(family_selected(AF_INET)))
3870         return;
3871
3872     /* 'for' loop 1: */
3873     for (; item; item = item->next_item) {
3874         if (Xflag) {
3875             (void) printf("\n--- Entry %d ---\n", ++jtemp);
3876             (void) printf("Group = %d, mib_id = %d, "
3877                 "length = %d, valp = 0x%p\n",
3878                 item->group, item->mib_id, item->length,
3879                 item->valp);
3880         }
3881         if (!(item->group == MIB2_IP && item->mib_id == MIB2_IP_MEDIA))
3882             continue; /* 'for' loop 1 */
3883
3884         if (Xflag)
3885             (void) printf("%u records for "
3886                 "ipNetToMediaEntryTable:\n",
3887                 item->length/sizeof (mib2_ipNetToMediaEntry_t));
3888
3889         first = B_TRUE;
3890         /* 'for' loop 2: */
3891         for (np = (mib2_ipNetToMediaEntry_t *)item->valp;

```

```

3892     (char *)np < (char *)item->valp + item->length;
3893     /* LINTED: (note 1) */
3894     np = (mib2_ipNetToMediaEntry_t *)((char *)np +
3895     ipNetToMediaEntrySize)) {
3896         if (first) {
3897             (void) puts(v4compat ?
3898                 "Net to Media Table" :
3899                 "Net to Media Table: IPv4");
3900             (void) puts("Device "
3901                 " IP Address          Mask  "
3902                 "Flags      Phys Addr");
3903             (void) puts("-----"
3904                 "-----");
3905             first = B_FALSE;
3906         }
3907
3908         flbuf[0] = '\0';
3909         flags = np->ipNetToMediaInfo.ntm_flags;
3910         /*
3911          * Note that not all flags are possible at the same
3912          * time. Patterns: SPIAy DUO
3913          */
3914         if (flags & ACE_F_PERMANENT)
3915             (void) strcat(flbuf, "S");
3916         if (flags & ACE_F_PUBLISH)
3917             (void) strcat(flbuf, "P");
3918         if (flags & ACE_F_DYING)
3919             (void) strcat(flbuf, "D");
3920         if (!(flags & ACE_F_RESOLVED))
3921             (void) strcat(flbuf, "U");
3922         if (flags & ACE_F_MAPPING)
3923             (void) strcat(flbuf, "M");
3924         if (flags & ACE_F_MYADDR)
3925             (void) strcat(flbuf, "L");
3926         if (flags & ACE_F_UNVERIFIED)
3927             (void) strcat(flbuf, "d");
3928         if (flags & ACE_F_AUTHORITY)
3929             (void) strcat(flbuf, "A");
3930         if (flags & ACE_F_OLD)
3931             (void) strcat(flbuf, "o");
3932         if (flags & ACE_F_DELAYED)
3933             (void) strcat(flbuf, "y");
3934         (void) printf("%-6s %-20s %-15s %-8s %s\n",
3935             octetstr(&np->ipNetToMediaIfIndex, 'a',
3936             ifname, sizeof (ifname)),
3937             pr_addr(np->ipNetToMediaNetAddress,
3938             abuf, sizeof (abuf)),
3939             octetstr(&np->ipNetToMediaInfo.ntm_mask, 'd',
3940             maskbuf, sizeof (maskbuf)),
3941             flbuf,
3942             octetstr(&np->ipNetToMediaPhysAddress, 'h',
3943             xbuf, sizeof (xbuf)));
3944     } /* 'for' loop 2 ends */
3945     } /* 'for' loop 1 ends */
3946     (void) fflush(stdout);
3947 }
3948
3950 /* ----- NDP_REPORT (netstat -p) ----- */
3951
3952 static void
3953 ndp_report(mib_item_t *item)
3954 {
3955     int             jtemp = 0;
3956     char             abuf[MAXHOSTNAMELEN + 1];
3957     char             *state;

```

```

3958     char          *type;
3959     char          xbuf[STR_EXPAND * OCTET_LENGTH + 1];
3960     mib2_ipNetToMediaEntry_t *np6;
3961     char          ifname[LIFNAMSIZ + 1];
3962     boolean_t     first;

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

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

3980         first = B_TRUE;
3981         /* 'for' loop 2: */
3982         for (np6 = (mib2_ipNetToMediaEntry_t *)item->valp;
3983             (char *)np6 < (char *)item->valp + item->length;
3984             /* LINTED: (note 1) */
3985             np6 = (mib2_ipNetToMediaEntry_t *)((char *)np6 +
3986             ipv6NetToMediaEntrySize)) {
3987             if (first) {
3988                 (void) puts("\nNet to Media Table: IPv6");
3989                 (void) puts(" If Physical Address "
3990                     " Type State Destination/Mask");
3991                 (void) puts("-----");
3992                 "-----";
3993                 "-----");
3994                 first = B_FALSE;
3995             }

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

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

```

```

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

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

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

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

4061 static void
4062 ire_report(const mib_item_t *item)
4063 {
4064     int jtemp = 0;
4065     boolean_t print_hdr_once_v4 = B_TRUE;
4066     boolean_t print_hdr_once_v6 = B_TRUE;
4067     mib2_ipRouteEntry_t *rp;
4068     mib2_ipRouteEntry_t *rp6;
4069     sec_attr_list_t **v4_attr, **v4a;
4070     sec_attr_list_t **v6_attr, **v6a;
4071     sec_attr_list_t *all_attr, *aptr;
4072     const mib_item_t *iptr;
4073     int ipv4_route_count, ipv6_route_count;
4074     int route_attr_count;

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

```

```

4090         ipRouteAttributeSize;
4091     }
4092     v4_attrs = v6_attrs = NULL;
4093     all_attrs = NULL;
4094     if (family_selected(AF_INET) && ipv4_route_count > 0) {
4095         v4_attrs = calloc(ipv4_route_count, sizeof(*v4_attrs));
4096         if (v4_attrs == NULL) {
4097             perror("ire_report calloc v4_attrs failed");
4098             return;
4099         }
4100     }
4101     if (family_selected(AF_INET6) && ipv6_route_count > 0) {
4102         v6_attrs = calloc(ipv6_route_count, sizeof(*v6_attrs));
4103         if (v6_attrs == NULL) {
4104             perror("ire_report calloc v6_attrs failed");
4105             goto ire_report_done;
4106         }
4107     }
4108     if (route_attrs_count > 0) {
4109         all_attrs = malloc(route_attrs_count * sizeof(*all_attrs));
4110         if (all_attrs == NULL) {
4111             perror("ire_report malloc all_attrs failed");
4112             goto ire_report_done;
4113         }
4114     }
4115     aptr = all_attrs;
4116     for (iptr = item; iptr != NULL; iptr = iptr->next_item) {
4117         mib2_ipAttributeEntry_t *iae;
4118         sec_attr_list_t **alp;
4119
4120         if (v4_attrs != NULL && iptr->group == MIB2_IP &&
4121             iptr->mib_id == EXPER_IP_RTATTR) {
4122             alp = v4_attrs;
4123         } else if (v6_attrs != NULL && iptr->group == MIB2_IP6 &&
4124             iptr->mib_id == EXPER_IP_RTATTR) {
4125             alp = v6_attrs;
4126         } else {
4127             continue;
4128         }
4129         for (iae = iptr->valp;
4130              (char *)iae < (char *)iptr->valp + iptr->length;
4131              /* LINTED: (note 1) */
4132              iae = (mib2_ipAttributeEntry_t *)((char *)iae +
4133                ipRouteAttributeSize)) {
4134             aptr->sal_next = alp[iae->iae_routeidx];
4135             aptr->sal_attr = iae;
4136             alp[iae->iae_routeidx] = aptr++;
4137         }
4138     }
4139
4140     /* 'for' loop 1: */
4141     v4a = v4_attrs;
4142     v6a = v6_attrs;
4143     for (; item != NULL; item = item->next_item) {
4144         if (Xflag) {
4145             (void) printf("\n--- Entry %d ---\n", ++jtemp);
4146             (void) printf("Group = %d, mib_id = %d, "
4147                "length = %d, valp = 0x%p\n",
4148                item->group, item->mib_id,
4149                item->length, item->valp);
4150         }
4151         if (!((item->group == MIB2_IP &&
4152             item->mib_id == MIB2_IP_ROUTE) ||
4153             (item->group == MIB2_IP6 &&
4154             item->mib_id == MIB2_IP6_ROUTE)))
4155             continue; /* 'for' loop 1 */

```

```

4157         if (item->group == MIB2_IP && !family_selected(AF_INET))
4158             continue; /* 'for' loop 1 */
4159         else if (item->group == MIB2_IP6 && !family_selected(AF_INET6))
4160             continue; /* 'for' loop 1 */
4161
4162         if (Xflag) {
4163             if (item->group == MIB2_IP) {
4164                 (void) printf("%u records for "
4165                    "ipRouteEntryTable:\n",
4166                    item->length/sizeof(mib2_ipRouteEntry_t));
4167             } else {
4168                 (void) printf("%u records for "
4169                    "ipv6RouteEntryTable:\n",
4170                    item->length/
4171                    sizeof(mib2_ipv6RouteEntry_t));
4172             }
4173         }
4174
4175         if (item->group == MIB2_IP) {
4176             for (rp = (mib2_ipRouteEntry_t *)item->valp;
4177                  (char *)rp < (char *)item->valp + item->length;
4178                  /* LINTED: (note 1) */
4179                  rp = (mib2_ipRouteEntry_t *)((char *)rp +
4180                    ipRouteEntrySize)) {
4181                 aptr = v4a == NULL ? NULL : *v4a++;
4182                 print_hdr_once_v4 = ire_report_item_v4(rp,
4183                    print_hdr_once_v4, aptr);
4184             }
4185         } else {
4186             for (rp6 = (mib2_ipv6RouteEntry_t *)item->valp;
4187                  (char *)rp6 < (char *)item->valp + item->length;
4188                  /* LINTED: (note 1) */
4189                  rp6 = (mib2_ipv6RouteEntry_t *)((char *)rp6 +
4190                    ipv6RouteEntrySize)) {
4191                 aptr = v6a == NULL ? NULL : *v6a++;
4192                 print_hdr_once_v6 = ire_report_item_v6(rp6,
4193                    print_hdr_once_v6, aptr);
4194             }
4195         }
4196     } /* 'for' loop 1 ends */
4197     (void) fflush(stdout);
4198     ire_report_done:
4199     if (v4_attrs != NULL)
4200         free(v4_attrs);
4201     if (v6_attrs != NULL)
4202         free(v6_attrs);
4203     if (all_attrs != NULL)
4204         free(all_attrs);
4205 }
4206
4207 /*
4208  * Match a user-supplied device name. We do this by string because
4209  * the MIB2 interface gives us interface name strings rather than
4210  * ifIndex numbers. The "none" rule matches only routes with no
4211  * interface. The "any" rule matches routes with any non-blank
4212  * interface. A base name ("hme0") matches all aliases as well
4213  * ("hme0:1").
4214  */
4215 static boolean_t
4216 dev_name_match(const DeviceName *devnam, const char *ifname)
4217 {
4218     int iflen;
4219
4220     if (ifname == NULL)
4221         return (devnam->o_length == 0); /* "none" */

```



```

4222     if (*ifname == '\0')
4223         return (devnam->o_length != 0);          /* "any" */
4224     iflen = strlen(ifname);
4225     /* The check for ':' here supports interface aliases. */
4226     if (iflen > devnam->o_length ||
4227         (iflen < devnam->o_length && devnam->o_bytes[iflen] != ':'))
4228         return (B_FALSE);
4229     return (strcmp(ifname, devnam->o_bytes, iflen) == 0);
4230 }
4231
4232 /*
4233  * Match a user-supplied IP address list. The "any" rule matches any
4234  * non-zero address. The "none" rule matches only the zero address.
4235  * IPv6 addresses supplied by the user are ignored. If the user
4236  * supplies a subnet mask, then match routes that are at least that
4237  * specific (use the user's mask). If the user supplies only an
4238  * address, then select any routes that would match (use the route's
4239  * mask).
4240  */
4241 static boolean_t
4242 v4_addr_match(IpAddress addr, IpAddress mask, const filter_t *fp)
4243 {
4244     char **app;
4245     char *aptr;
4246     in_addr_t faddr, fmask;
4247
4248     if (fp->u.a.f_address == NULL) {
4249         if (IN6_IS_ADDR_UNSPECIFIED(&fp->u.a.f_mask))
4250             return (addr != INADDR_ANY);        /* "any" */
4251         else
4252             return (addr == INADDR_ANY);        /* "none" */
4253     }
4254     if (!IN6_IS_V4MASK(fp->u.a.f_mask))
4255         return (B_FALSE);
4256     IN6_V4MAPPED_TO_IPADDR(&fp->u.a.f_mask, fmask);
4257     if (fmask != IP_HOST_MASK) {
4258         if (fmask > mask)
4259             return (B_FALSE);
4260         mask = fmask;
4261     }
4262     for (app = fp->u.a.f_address->h_addr_list; (aptr = *app) != NULL; app++)
4263         /* LINTED: (note 1) */
4264         if (IN6_IS_ADDR_V4MAPPED((in6_addr_t *)aptr)) {
4265             /* LINTED: (note 1) */
4266             IN6_V4MAPPED_TO_IPADDR((in6_addr_t *)aptr, faddr);
4267             if (((faddr ^ addr) & mask) == 0)
4268                 return (B_TRUE);
4269         }
4270     return (B_FALSE);
4271 }
4272
4273 /*
4274  * Run through the filter list for an IPv4 MIB2 route entry. If all
4275  * filters of a given type fail to match, then the route is filtered
4276  * out (not displayed). If no filter is given or at least one filter
4277  * of each type matches, then display the route.
4278  */
4279 static boolean_t
4280 ire_filter_match_v4(const mib2_ipRouteEntry_t *rp, uint_t flag_b)
4281 {
4282     filter_t *fp;
4283     int idx;
4284
4285     /* 'for' loop 1: */
4286     for (idx = 0; idx < NFILTERKEYS; idx++)
4287         if ((fp = filters[idx]) != NULL) {

```

```

4288         /* 'for' loop 2: */
4289         for (; fp != NULL; fp = fp->f_next) {
4290             switch (idx) {
4291                 case FK_AF:
4292                     if (fp->u.f_family != AF_INET)
4293                         continue; /* 'for' loop 2 */
4294                     break;
4295                 case FK_OUTIF:
4296                     if (!dev_name_match(&rp->ipRouteIfIndex,
4297                                         fp->u.f_ifname))
4298                         continue; /* 'for' loop 2 */
4299                     break;
4300                 case FK_DST:
4301                     if (!v4_addr_match(rp->ipRouteDest,
4302                                         rp->ipRouteMask, fp))
4303                         continue; /* 'for' loop 2 */
4304                     break;
4305                 case FK_FLAGS:
4306                     if ((flag_b & fp->u.f.f_flagset) !=
4307                         fp->u.f.f_flagset ||
4308                         (flag_b & fp->u.f.f_flagclear))
4309                         continue; /* 'for' loop 2 */
4310                     break;
4311             }
4312             /* 'for' loop 2 ends */
4313             if (fp == NULL)
4314                 return (B_FALSE);
4315         }
4316     /* 'for' loop 1 ends */
4317     return (B_TRUE);
4318 }
4319
4320 /*
4321  * Given an IPv4 MIB2 route entry, form the list of flags for the
4322  * route.
4323  */
4324 static uint_t
4325 form_v4_route_flags(const mib2_ipRouteEntry_t *rp, char *flags)
4326 {
4327     uint_t flag_b;
4328
4329     flag_b = FLF_U;
4330     (void) strcpy(flags, "U");
4331     /* RTF_INDIRECT wins over RTF_GATEWAY - don't display both */
4332     if (rp->ipRouteInfo.re_flags & RTF_INDIRECT) {
4333         (void) strcat(flags, "I");
4334         flag_b |= FLF_I;
4335     } else if (rp->ipRouteInfo.re_ire_type & IRE_OFFLINK) {
4336         (void) strcat(flags, "G");
4337         flag_b |= FLF_G;
4338     }
4339     /* IRE_IF_CLONE wins over RTF_HOST - don't display both */
4340     if (rp->ipRouteInfo.re_ire_type & IRE_IF_CLONE) {
4341         (void) strcat(flags, "C");
4342         flag_b |= FLF_C;
4343     } else if (rp->ipRouteMask == IP_HOST_MASK) {
4344         (void) strcat(flags, "H");
4345         flag_b |= FLF_H;
4346     }
4347     if (rp->ipRouteInfo.re_flags & RTF_DYNAMIC) {
4348         (void) strcat(flags, "D");
4349         flag_b |= FLF_D;
4350     }
4351     if (rp->ipRouteInfo.re_ire_type == IRE_BROADCAST) { /* Broadcast */
4352         (void) strcat(flags, "b");
4353     }

```

```

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

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

4393 static const char ire_hdr_v4_normal[] =
4394     " Destination      Gateway      Flags Ref Use Interface"
4395     "%s\n"
4396     "-----"
4397     "%s\n";

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

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

4417     flag_b = form_v4_route_flags(rp, flags);

4419     if (!ire_filter_match_v4(rp, flag_b))

```

```

4420         return (first);

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

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

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

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

```

```

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

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

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

```

```

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

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

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

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

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

```

```

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

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

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

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

4664     flag_b = form_v6_route_flags(rp6, flags);

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

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

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

```

```

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

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

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

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

```

```

4750         return;
4752         if (bisinvalid(&attr->tme_label)) {
4753             (void) printf("  INVALID\n");
4754         } else if (!blequal(&attr->tme_label, zone_security_label)) {
4755             char *sl_str;
4757             sl_str = sl_to_str(&attr->tme_label);
4758             (void) printf("    %s\n", sl_str);
4759             free(sl_str);
4760         }
4761     }
4763 /* ----- TCP_REPORT----- */
4765 static const char tcp_hdr_v4[] =
4766     "\nTCP: IPv4\n";
4767 static const char tcp_hdr_v4_compat[] =
4768     "\nTCP\n";
4769 static const char tcp_hdr_v4_verbose[] =
4770     "Local/Remote Address Swind Snext Suna Rwind Rnext Rack "
4771     "Rto Mss State\n";
4772 "-----\n";
4773 "-----\n";
4774 static const char tcp_hdr_v4_normal[] =
4775     " Local Address Remote Address Swind Send-Q Rwind Recv-Q "
4776     " State\n";
4777 "-----\n";
4778 "-----\n";
4780 static const char tcp_hdr_v6[] =
4781     "\nTCP: IPv6\n";
4782 static const char tcp_hdr_v6_verbose[] =
4783     "Local/Remote Address Swind Snext Suna Rwind Rnext "
4784     " Rack Rto Mss State If\n";
4785 "-----\n";
4786 "-----\n";
4787 static const char tcp_hdr_v6_normal[] =
4788     " Local Address Remote Address "
4789     "Swind Send-Q Rwind Recv-Q State If\n";
4790 "-----\n";
4791 "-----\n";
4793 static boolean_t tcp_report_item_v4(const mib2_tcpConnEntry_t *,
4794     boolean_t first, const mib2_transportMLPEntry_t *);
4795 static boolean_t tcp_report_item_v6(const mib2_tcp6ConnEntry_t *,
4796     boolean_t first, const mib2_transportMLPEntry_t *);
4798 static void
4799 tcp_report(const mib_item_t *item)
4800 {
4801     int jtemp = 0;
4802     boolean_t print_hdr_once_v4 = B_TRUE;
4803     boolean_t print_hdr_once_v6 = B_TRUE;
4804     mib2_tcpConnEntry_t *tp;
4805     mib2_tcp6ConnEntry_t *tp6;
4806     mib2_transportMLPEntry_t **v4_attrs, **v6_attrs;
4807     mib2_transportMLPEntry_t **v4a, **v6a;
4808     mib2_transportMLPEntry_t *aptr;
4810     if (!protocol_selected(IPPROTO_TCP))
4811         return;
4813 /*
4814  * Preparation pass: the kernel returns separate entries for TCP
4815  * connection table entries and Multilevel Port attributes. We loop

```

```

4816     * through the attributes first and set up an array for each address
4817     * family.
4818     */
4819     v4_attrs = family_selected(AF_INET) && RSECflag ?
4820         gather_attrs(item, MIB2_TCP, MIB2_TCP_CONN, tcpConnEntrySize) :
4821         NULL;
4822     v6_attrs = family_selected(AF_INET6) && RSECflag ?
4823         gather_attrs(item, MIB2_TCP6, MIB2_TCP6_CONN, tcp6ConnEntrySize) :
4824         NULL;
4826     /* 'for' loop 1: */
4827     v4a = v4_attrs;
4828     v6a = v6_attrs;
4829     for (; item != NULL; item = item->next_item) {
4830         if (Xflag) {
4831             (void) printf("\n--- Entry %d ---\n", ++jtemp);
4832             (void) printf("Group = %d, mib_id = %d, "
4833                 "length = %d, valp = 0x%p\n",
4834                 item->group, item->mib_id,
4835                 item->length, item->valp);
4836         }
4838         if (!((item->group == MIB2_TCP &&
4839             item->mib_id == MIB2_TCP_CONN) ||
4840             (item->group == MIB2_TCP6 &&
4841             item->mib_id == MIB2_TCP6_CONN)))
4842             continue; /* 'for' loop 1 */
4844         if (item->group == MIB2_TCP && !family_selected(AF_INET))
4845             continue; /* 'for' loop 1 */
4846         else if (item->group == MIB2_TCP6 && !family_selected(AF_INET6))
4847             continue; /* 'for' loop 1 */
4849         if (item->group == MIB2_TCP) {
4850             for (tp = (mib2_tcpConnEntry_t *)item->valp;
4851                 (char *)tp < (char *)item->valp + item->length;
4852                 /* LINTED: (note 1) */
4853                 tp = (mib2_tcpConnEntry_t *)((char *)tp +
4854                     tcpConnEntrySize)) {
4855                 aptr = v4a == NULL ? NULL : *v4a++;
4856                 print_hdr_once_v4 = tcp_report_item_v4(tp,
4857                     print_hdr_once_v4, aptr);
4858             }
4859         } else {
4860             for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4861                 (char *)tp6 < (char *)item->valp + item->length;
4862                 /* LINTED: (note 1) */
4863                 tp6 = (mib2_tcp6ConnEntry_t *)((char *)tp6 +
4864                     tcp6ConnEntrySize)) {
4865                 aptr = v6a == NULL ? NULL : *v6a++;
4866                 print_hdr_once_v6 = tcp_report_item_v6(tp6,
4867                     print_hdr_once_v6, aptr);
4868             }
4869         }
4870     } /* 'for' loop 1 ends */
4871     (void) fflush(stdout);
4873     if (v4_attrs != NULL)
4874         free(v4_attrs);
4875     if (v6_attrs != NULL)
4876         free(v6_attrs);
4877 }
4879 static boolean_t
4880 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, boolean_t first,
4881     const mib2_transportMLPEntry_t *attr)

```

```

4882 {
4883     /*
4884      * lname and fname below are for the hostname as well as the portname
4885      * There is no limit on portname length so we assume MAXHOSTNAMELEN
4886      * as the limit
4887      */
4888     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4889     char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4890
4891     if (!(Aflag || tp->tcpConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4892         return (first); /* Nothing to print */
4893
4894     if (first) {
4895         (void) printf(v4compat ? tcp_hdr_v4_compat : tcp_hdr_v4);
4896         (void) printf(Vflag ? tcp_hdr_v4_verbose : tcp_hdr_v4_normal);
4897     }
4898
4899     if (Vflag) {
4900         (void) printf("%-20s\n%-20s %5u %08x %08x %5u %08x %08x "
4901             "%5u %5u %s\n",
4902             pr_ap(tp->tcpConnLocalAddress,
4903                 tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4904             pr_ap(tp->tcpConnRemAddress,
4905                 tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4906             tp->tcpConnEntryInfo.ce_swnd,
4907             tp->tcpConnEntryInfo.ce_snxt,
4908             tp->tcpConnEntryInfo.ce_suna,
4909             tp->tcpConnEntryInfo.ce_rwnd,
4910             tp->tcpConnEntryInfo.ce_rnxt,
4911             tp->tcpConnEntryInfo.ce_rack,
4912             tp->tcpConnEntryInfo.ce_rto,
4913             tp->tcpConnEntryInfo.ce_mss,
4914             mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4915     } else {
4916         int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
4917             (int)tp->tcpConnEntryInfo.ce_suna - 1;
4918         int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
4919             (int)tp->tcpConnEntryInfo.ce_rack;
4920
4921         (void) printf("%-20s %-20s %5u %6d %5u %6d %s\n",
4922             pr_ap(tp->tcpConnLocalAddress,
4923                 tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4924             pr_ap(tp->tcpConnRemAddress,
4925                 tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4926             tp->tcpConnEntryInfo.ce_swnd,
4927             (sq >= 0) ? sq : 0,
4928             tp->tcpConnEntryInfo.ce_rwnd,
4929             (rq >= 0) ? rq : 0,
4930             mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4931     }
4932
4933     print_transport_label(attr);
4934
4935     return (B_FALSE);
4936 }
4937
4938 static boolean_t
4939 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, boolean_t first,
4940     const mib2_transportMLPEntry_t *attr)
4941 {
4942     /*
4943      * lname and fname below are for the hostname as well as the portname
4944      * There is no limit on portname length so we assume MAXHOSTNAMELEN
4945      * as the limit
4946      */
4947     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];

```

```

4948     char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4949     char    ifname[LIFNAMSIZ + 1];
4950     char    *ifnamep;
4951
4952     if (!(Aflag || tp6->tcp6ConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4953         return (first); /* Nothing to print */
4954
4955     if (first) {
4956         (void) printf(tcp_hdr_v6);
4957         (void) printf(Vflag ? tcp_hdr_v6_verbose : tcp_hdr_v6_normal);
4958     }
4959
4960     ifnamep = (tp6->tcp6ConnIfIndex != 0) ?
4961         if_indextoname(tp6->tcp6ConnIfIndex, ifname) : NULL;
4962     if (ifnamep == NULL)
4963         ifnamep = "";
4964
4965     if (Vflag) {
4966         (void) printf("%-33s\n%-33s %5u %08x %08x %5u %08x %08x "
4967             "%5u %5u %-11s %s\n",
4968             pr_ap6(&tp6->tcp6ConnLocalAddress,
4969                 tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
4970             pr_ap6(&tp6->tcp6ConnRemAddress,
4971                 tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
4972             tp6->tcp6ConnEntryInfo.ce_swnd,
4973             tp6->tcp6ConnEntryInfo.ce_snxt,
4974             tp6->tcp6ConnEntryInfo.ce_suna,
4975             tp6->tcp6ConnEntryInfo.ce_rwnd,
4976             tp6->tcp6ConnEntryInfo.ce_rnxt,
4977             tp6->tcp6ConnEntryInfo.ce_rack,
4978             tp6->tcp6ConnEntryInfo.ce_rto,
4979             tp6->tcp6ConnEntryInfo.ce_mss,
4980             mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
4981             ifnamep);
4982     } else {
4983         int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
4984             (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
4985         int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
4986             (int)tp6->tcp6ConnEntryInfo.ce_rack;
4987
4988         (void) printf("%-33s %-33s %5u %6d %5u %6d %-11s %s\n",
4989             pr_ap6(&tp6->tcp6ConnLocalAddress,
4990                 tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
4991             pr_ap6(&tp6->tcp6ConnRemAddress,
4992                 tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
4993             tp6->tcp6ConnEntryInfo.ce_swnd,
4994             (sq >= 0) ? sq : 0,
4995             tp6->tcp6ConnEntryInfo.ce_rwnd,
4996             (rq >= 0) ? rq : 0,
4997             mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
4998             ifnamep);
4999     }
5000
5001     print_transport_label(attr);
5002
5003     return (B_FALSE);
5004 }
5005
5006 /* ----- UDP_REPORT----- */
5007
5008 static boolean_t udp_report_item_v4(const mib2_udpEntry_t *ude,
5009     boolean_t first, const mib2_transportMLPEntry_t *attr);
5010 static boolean_t udp_report_item_v6(const mib2_udp6Entry_t *ude6,
5011     boolean_t first, const mib2_transportMLPEntry_t *attr);
5012
5013 static const char udp_hdr_v4[] =

```

```

5014 "      Local Address      Remote Address      State\n"
5015 "-----\n";

5017 static const char udp_hdr_v6[] =
5018 "      Local Address      Remote Address      "
5019 "      State      If\n"
5020 "-----\n"
5021 "-----\n";

5023 static void
5024 udp_report(const mib_item_t *item)
5025 {
5026     int                jtemp = 0;
5027     boolean_t          print_hdr_once_v4 = B_TRUE;
5028     boolean_t          print_hdr_once_v6 = B_TRUE;
5029     mib2_udpEntry_t    *ude;
5030     mib2_udp6Entry_t   *ude6;
5031     mib2_transportMLPEntry_t **v4_attrs, **v6_attrs;
5032     mib2_transportMLPEntry_t **v4a, **v6a;
5033     mib2_transportMLPEntry_t *aptr;

5035     if (!protocol_selected(IPPROTO_UDP))
5036         return;

5038     /*
5039     * Preparation pass: the kernel returns separate entries for UDP
5040     * connection table entries and Multilevel Port attributes. We loop
5041     * through the attributes first and set up an array for each address
5042     * family.
5043     */
5044     v4_attrs = family_selected(AF_INET) && RSECflag ?
5045         gather_attrs(item, MIB2_UDP, MIB2_UDP_ENTRY, udpEntrySize) : NULL;
5046     v6_attrs = family_selected(AF_INET6) && RSECflag ?
5047         gather_attrs(item, MIB2_UDP6, MIB2_UDP6_ENTRY, udp6EntrySize) :
5048         NULL;

5050     v4a = v4_attrs;
5051     v6a = v6_attrs;
5052     /* 'for' loop 1: */
5053     for (; item; item = item->next_item) {
5054         if (Xflag) {
5055             (void) printf("\n-- Entry %d ---\n", ++jtemp);
5056             (void) printf("Group = %d, mib_id = %d, "
5057                 "length = %d, valp = 0x%p\n",
5058                 item->group, item->mib_id,
5059                 item->length, item->valp);
5060         }
5061         if (((item->group == MIB2_UDP &&
5062             item->mib_id == MIB2_UDP_ENTRY) ||
5063             (item->group == MIB2_UDP6 &&
5064             item->mib_id == MIB2_UDP6_ENTRY)))
5065             continue; /* 'for' loop 1 */

5067         if (item->group == MIB2_UDP && !family_selected(AF_INET))
5068             continue; /* 'for' loop 1 */
5069         else if (item->group == MIB2_UDP6 && !family_selected(AF_INET6))
5070             continue; /* 'for' loop 1 */

5072         /*      xxx.xxx.xxx.xxx,pppp sss... */
5073         if (item->group == MIB2_UDP) {
5074             for (ude = (mib2_udpEntry_t *)item->valp;
5075                 (char *)ude < (char *)item->valp + item->length;
5076                 /* LINTED: (note 1) */
5077                 ude = (mib2_udpEntry_t *)((char *)ude +
5078                     udpEntrySize)) {
5079                 aptr = v4a == NULL ? NULL : *v4a++;

```

```

5080                 print_hdr_once_v4 = udp_report_item_v4(ude,
5081                     print_hdr_once_v4, aptr);
5082             } else {
5083                 for (ude6 = (mib2_udp6Entry_t *)item->valp;
5084                     (char *)ude6 < (char *)item->valp + item->length;
5085                     /* LINTED: (note 1) */
5086                     ude6 = (mib2_udp6Entry_t *)((char *)ude6 +
5087                         udp6EntrySize)) {
5088                     aptr = v6a == NULL ? NULL : *v6a++;
5089                     print_hdr_once_v6 = udp_report_item_v6(ude6,
5090                         print_hdr_once_v6, aptr);
5091                 }
5092             }
5093         } /* 'for' loop 1 ends */
5094         (void) fflush(stdout);
5095     }

5097     if (v4_attrs != NULL)
5098         free(v4_attrs);
5099     if (v6_attrs != NULL)
5100         free(v6_attrs);
5101 }

5103 static boolean_t
5104 udp_report_item_v4(const mib2_udpEntry_t *ude, boolean_t first,
5105     const mib2_transportMLPEntry_t *attr)
5106 {
5107     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5108           /* hostname + portname */

5110     if (!(Aflag || ude->udpEntryInfo.ue_state >= MIB2_UDP_connected))
5111         return (first); /* Nothing to print */

5113     if (first) {
5114         (void) printf(v4compat ? "\nUDP\n" : "\nUDP: IPv4\n");
5115         (void) printf(udp_hdr_v4);
5116         first = B_FALSE;
5117     }

5119     (void) printf("%-20s ",
5120         pr_ap(ude->udpLocalAddress, ude->udpLocalPort, "udp",
5121             lname, sizeof (lname)));
5122     (void) printf("%-20s %s\n",
5123         ude->udpEntryInfo.ue_state == MIB2_UDP_connected ?
5124         pr_ap(ude->udpEntryInfo.ue_RemoteAddress,
5125             ude->udpEntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5126         "",
5127         miudp_state(ude->udpEntryInfo.ue_state, attr));

5129     print_transport_label(attr);

5131     return (first);
5132 }

5134 static boolean_t
5135 udp_report_item_v6(const mib2_udp6Entry_t *ude6, boolean_t first,
5136     const mib2_transportMLPEntry_t *attr)
5137 {
5138     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5139           /* hostname + portname */
5140     char    ifname[LIFNAMSIZ + 1];
5141     const char *ifnamep;

5143     if (!(Aflag || ude6->udp6EntryInfo.ue_state >= MIB2_UDP_connected))
5144         return (first); /* Nothing to print */

```

```

5146     if (first) {
5147         (void) printf("\nUDP: IPv6\n");
5148         (void) printf(udp_hdr_v6);
5149         first = B_FALSE;
5150     }
5152     ifnamep = (ude6->udp6IfIndex != 0) ?
5153         if_indextoname(ude6->udp6IfIndex, ifname) : NULL;
5155     (void) printf("%-33s ",
5156         pr_ap6(&ude6->udp6LocalAddress,
5157         ude6->udp6LocalPort, "udp", lname, sizeof (lname)));
5158     (void) printf("%-33s %-10s %s\n",
5159         ude6->udp6EntryInfo.ue_state == MIB2_UDP_connected ?
5160         pr_ap6(&ude6->udp6EntryInfo.ue_RemoteAddress,
5161         ude6->udp6EntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5162         "",
5163         miudp_state(ude6->udp6EntryInfo.ue_state, attr),
5164         ifnamep == NULL ? "" : ifnamep);
5166     print_transport_label(attr);
5168     return (first);
5169 }
5171 /* ----- SCTP_REPORT----- */
5173 static const char sctp_hdr[] =
5174     "\nSCTP:";
5175 static const char sctp_hdr_normal[] =
5176     "      Local Address      Remote Address      "
5177     "Swind Send-Q Rwind Recv-Q StrsI/O State\n"
5178     "-----"
5179     "-----";
5181 static const char *
5182 nssctp_state(int state, const mib2_transportMLPEntry_t *attr)
5183 {
5184     static char sctpsbuf[50];
5185     const char *cp;
5187     switch (state) {
5188     case MIB2_SCTP_closed:
5189         cp = "CLOSED";
5190         break;
5191     case MIB2_SCTP_cookieWait:
5192         cp = "COOKIE_WAIT";
5193         break;
5194     case MIB2_SCTP_cookieEchoed:
5195         cp = "COOKIE_ECHOED";
5196         break;
5197     case MIB2_SCTP_established:
5198         cp = "ESTABLISHED";
5199         break;
5200     case MIB2_SCTP_shutdownPending:
5201         cp = "SHUTDOWN_PENDING";
5202         break;
5203     case MIB2_SCTP_shutdownSent:
5204         cp = "SHUTDOWN_SENT";
5205         break;
5206     case MIB2_SCTP_shutdownReceived:
5207         cp = "SHUTDOWN_RECEIVED";
5208         break;
5209     case MIB2_SCTP_shutdownAckSent:
5210         cp = "SHUTDOWN_ACK_SENT";
5211         break;

```

```

5212     case MIB2_SCTP_listen:
5213         cp = "LISTEN";
5214         break;
5215     default:
5216         (void) snprintf(sctpsbuf, sizeof (sctpsbuf),
5217             "UNKNOWN STATE(%d)", state);
5218         cp = sctpsbuf;
5219         break;
5220     }
5222     if (RSECflag && attr != NULL && attr->tme_flags != 0) {
5223         if (cp != sctpsbuf) {
5224             (void) strlcpy(sctpsbuf, cp, sizeof (sctpsbuf));
5225             cp = sctpsbuf;
5226         }
5227         if (attr->tme_flags & MIB2_TMEF_PRIVATE)
5228             (void) strlcat(sctpsbuf, " P", sizeof (sctpsbuf));
5229         if (attr->tme_flags & MIB2_TMEF_SHARED)
5230             (void) strlcat(sctpsbuf, " S", sizeof (sctpsbuf));
5231     }
5233     return (cp);
5234 }
5236 static const mib2_sctpConnRemoteEntry_t *
5237 sctp_getnext_rem(const mib_item_t **itemp,
5238     const mib2_sctpConnRemoteEntry_t *current, uint32_t associd)
5239 {
5240     const mib_item_t *item = *itemp;
5241     const mib2_sctpConnRemoteEntry_t *sre;
5243     for (; item != NULL; item = item->next_item, current = NULL) {
5244         if (!(item->group == MIB2_SCTP &&
5245             item->mib_id == MIB2_SCTP_CONN_REMOTE)) {
5246             continue;
5247         }
5249         if (current != NULL) {
5250             /* LINTED: (note 1) */
5251             sre = (const mib2_sctpConnRemoteEntry_t *)
5252                 ((const char *)current + sctpRemoteEntrySize);
5253         } else {
5254             sre = item->valp;
5255         }
5256         for (; (char *)sre < (char *)item->valp + item->length;
5257             /* LINTED: (note 1) */
5258             sre = (const mib2_sctpConnRemoteEntry_t *)
5259                 ((const char *)sre + sctpRemoteEntrySize)) {
5260             if (sre->sctpAssocId != associd) {
5261                 continue;
5262             }
5263             *itemp = item;
5264             return (sre);
5265         }
5266     }
5267     *itemp = NULL;
5268     return (NULL);
5269 }
5271 static const mib2_sctpConnLocalEntry_t *
5272 sctp_getnext_local(const mib_item_t **itemp,
5273     const mib2_sctpConnLocalEntry_t *current, uint32_t associd)
5274 {
5275     const mib_item_t *item = *itemp;
5276     const mib2_sctpConnLocalEntry_t *sle;

```



```

5278     for (; item != NULL; item = item->next_item, current = NULL) {
5279         if (!(item->group == MIB2_SCTP &&
5280             item->mib_id == MIB2_SCTP_CONN_LOCAL)) {
5281             continue;
5282         }

5284         if (current != NULL) {
5285             /* LINTED: (note 1) */
5286             sle = (const mib2_sctpConnLocalEntry_t *)
5287                 ((const char *)current + sctpLocalEntrySize);
5288         } else {
5289             sle = item->valp;
5290         }
5291         for (; (char *)sle < (char *)item->valp + item->length;
5292             /* LINTED: (note 1) */
5293             sle = (const mib2_sctpConnLocalEntry_t *)
5294                 ((const char *)sle + sctpLocalEntrySize)) {
5295             if (sle->sctpAssocId != associd) {
5296                 continue;
5297             }
5298             *itemp = item;
5299             return (sle);
5300         }
5301     }
5302     *itemp = NULL;
5303     return (NULL);
5304 }

5306 static void
5307 sctp_pr_addr(int type, char *name, int namelen, const in6_addr_t *addr,
5308             int port)
5309 {
5310     ipaddr_t      v4addr;
5311     in6_addr_t    v6addr;

5313     /*
5314      * Address is either a v4 mapped or v6 addr. If
5315      * it's a v4 mapped, convert to v4 before
5316      * displaying.
5317      */
5318     switch (type) {
5319     case MIB2_SCTP_ADDR_V4:
5320         /* v4 */
5321         v6addr = *addr;

5323         IN6_V4MAPPED_TO_IPADDR(&v4addr, v6addr);
5324         if (port > 0) {
5325             (void) pr_ap(v4addr, port, "sctp", name, namelen);
5326         } else {
5327             (void) pr_addr(v4addr, name, namelen);
5328         }
5329         break;

5331     case MIB2_SCTP_ADDR_V6:
5332         /* v6 */
5333         if (port > 0) {
5334             (void) pr_ap6(addr, port, "sctp", name, namelen);
5335         } else {
5336             (void) pr_addr6(addr, name, namelen);
5337         }
5338         break;

5340     default:
5341         (void) snprintf(name, namelen, "<unknown addr type>");
5342         break;
5343 }

```

```

5344 }

5346 static void
5347 sctp_conn_report_item(const mib_item_t *head, const mib2_sctpConnEntry_t *sp,
5348                     const mib2_transportMLPEntry_t *attr)
5349 {
5350     char          lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5351     char          fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5352     const mib2_sctpConnRemoteEntry_t *sre = NULL;
5353     const mib2_sctpConnLocalEntry_t *sle = NULL;
5354     const mib_item_t *local = head;
5355     const mib_item_t *remote = head;
5356     uint32_t      id = sp->sctpAssocId;
5357     boolean_t      printfirst = B_TRUE;

5359     sctp_pr_addr(sp->sctpAssocRemPrimAddrType, fname, sizeof (fname),
5360                 &sp->sctpAssocRemPrimAddr, sp->sctpAssocRemPort);
5361     sctp_pr_addr(sp->sctpAssocRemPrimAddrType, lname, sizeof (lname),
5362                 &sp->sctpAssocLocPrimAddr, sp->sctpAssocLocalPort);

5364     (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %s\n",
5365                 lname, fname,
5366                 sp->sctpConnEntryInfo.ce_swnd,
5367                 sp->sctpConnEntryInfo.ce_sendq,
5368                 sp->sctpConnEntryInfo.ce_rwnd,
5369                 sp->sctpConnEntryInfo.ce_recvq,
5370                 sp->sctpAssocInStreams, sp->sctpAssocOutStreams,
5371                 nssctp_state(sp->sctpAssocState, attr));

5373     print_transport_label(attr);

5375     if (!Vflag) {
5376         return;
5377     }

5379     /* Print remote addresses/local addresses on following lines */
5380     while ((sre = sctp_getnext_rem(&remote, sre, id)) != NULL) {
5381         if (!IN6_ARE_ADDR_EQUAL(&sre->sctpAssocRemAddr,
5382                                 &sp->sctpAssocRemPrimAddr)) {
5383             if (printfirst == B_TRUE) {
5384                 (void) fputs("\t<Remote: ", stdout);
5385                 printfirst = B_FALSE;
5386             } else {
5387                 (void) fputs(", ", stdout);
5388             }
5389             sctp_pr_addr(sre->sctpAssocRemAddrType, fname,
5390                         sizeof (fname), &sre->sctpAssocRemAddr, -1);
5391             if (sre->sctpAssocRemAddrActive == MIB2_SCTP_ACTIVE) {
5392                 (void) fputs(fname, stdout);
5393             } else {
5394                 (void) printf("(%s)", fname);
5395             }
5396         }
5397     }
5398     if (printfirst == B_FALSE) {
5399         (void) puts(">");
5400         printfirst = B_TRUE;
5401     }
5402     while ((sle = sctp_getnext_local(&local, sle, id)) != NULL) {
5403         if (!IN6_ARE_ADDR_EQUAL(&sle->sctpAssocLocalAddr,
5404                                 &sp->sctpAssocLocPrimAddr)) {
5405             if (printfirst == B_TRUE) {
5406                 (void) fputs("\t<Local: ", stdout);
5407                 printfirst = B_FALSE;
5408             } else {
5409                 (void) fputs(", ", stdout);

```

```

5410     }
5411     sctp_pr_addr(sle->sctpAssocLocalAddrType, lname,
5412               sizeof (lname), &sle->sctpAssocLocalAddr, -1);
5413     (void) fputs(lname, stdout);
5414 }
5415 }
5416 if (printfirst == B_FALSE) {
5417     (void) puts(">");
5418 }
5419 }

5421 static void
5422 sctp_report(const mib_item_t *item)
5423 {
5424     const mib_item_t      *head;
5425     const mib2_sctpConnEntry_t *sp;
5426     boolean_t             first = B_TRUE;
5427     mib2_transportMLPEntry_t **attrs, **aptr;
5428     mib2_transportMLPEntry_t *attr;

5430     /*
5431      * Preparation pass: the kernel returns separate entries for SCTP
5432      * connection table entries and Multilevel Port attributes. We loop
5433      * through the attributes first and set up an array for each address
5434      * family.
5435      */
5436     attrs = RSECflag ?
5437         gather_attrs(item, MIB2_SCTP, MIB2_SCTP_CONN, sctpEntrySize) :
5438         NULL;

5440     aptr = attrs;
5441     head = item;
5442     for (; item != NULL; item = item->next_item) {

5444         if (!(item->group == MIB2_SCTP &&
5445             item->mib_id == MIB2_SCTP_CONN))
5446             continue;

5448         for (sp = item->valp;
5449              (char *)sp < (char *)item->valp + item->length;
5450              /* LINTED: (note 1) */
5451              sp = (mib2_sctpConnEntry_t *)((char *)sp + sctpEntrySize)) {
5452             attr = aptr == NULL ? NULL : *aptr++;
5453             if (Aflag ||
5454                 sp->sctpAssocState >= MIB2_SCTP_established) {
5455                 if (first == B_TRUE) {
5456                     (void) puts(sctp_hdr);
5457                     (void) puts(sctp_hdr_normal);
5458                     first = B_FALSE;
5459                 }
5460                 sctp_conn_report_item(head, sp, attr);
5461             }
5462         }
5463     }
5464     if (attrs != NULL)
5465         free(attrs);
5466 }

5468 /* ----- DCCP_REPORT----- */

5470 static const char dccp_hdr_v4[] =
5471     "\nDCCP: IPv4\n";
5472 static const char dccp_hdr_v4_compat[] =
5473     "\nDCCP\n";
5474 static const char dccp_hdr_v4_verbose[] =
5475     "Local/Remote Address Swind Snext Suna Rwind Rnext Rack "
```

```

5476     "Rto Mss State\n"
5477     "-----\n";
5478     static const char dccp_hdr_v4_normal[] =
5479     " Local Address Remote Address Swind Send-Q Rwind Recv-Q "
5480     " State\n"
5481     "-----\n";
5482     static const char dccp_hdr_v6[] =
5483     "\nDCCP: IPv6\n";
5484     static const char dccp_hdr_v6_verbose[] =
5485     "Local/Remote Address Swind Snext Suna Rwind Rnext "
5486     " Rack Rto Mss State If\n"
5487     "-----\n";
5488     static const char dccp_hdr_v6_normal[] =
5489     " Local Address Remote Address "
5490     " Swind Send-Q Rwind Recv-Q State If\n"
5491     "-----\n";

5498 static boolean_t dccp_report_item_v4(const mib2_dccpConnEntry_t *,
5499 boolean_t, const mib2_transportMLPEntry_t *);
5500 static boolean_t dccp_report_item_v6(const mib2_dccp6ConnEntry_t *,
5501 boolean_t, const mib2_transportMLPEntry_t *);

5503 static void
5504 dccp_report(const mib_item_t *item)
5505 {
5506     mib2_dccpConnEntry_t *dp;
5507     mib2_transportMLPEntry_t **v4_attrs;
5508     mib2_transportMLPEntry_t **v6_attrs;
5509     mib2_transportMLPEntry_t **v4a;
5510     mib2_transportMLPEntry_t **v6a;
5511     mib2_transportMLPEntry_t *aptr;
5512     boolean_t print_hdr_once_v4 = B_TRUE;
5513     boolean_t print_hdr_once_v6 = B_TRUE;
5514     int jtemp = 0;

5516     if (!protocol_selected(IPPROTO_DCCP)) {
5517         return;
5518     }

5520     v4_attrs = family_selected(AF_INET) && RSECflag ?
5521         gather_attrs(item, MIB2_DCCP, MIB2_DCCP_CONN, dccpEntrySize) :
5522         NULL;
5523     v6_attrs = family_selected(AF_INET6) && RSECflag ?
5524         gather_attrs(item, MIB2_DCCP6, MIB2_DCCP6_CONN, dccp6EntrySize) :
5525         NULL;

5527     v4a = v4_attrs;
5528     v6a = v6_attrs;
5529     for (; item != NULL; item = item->next_item) {
5530         if (Xflag) {
5531             (void) printf("\n--- Entry %d ---\n", ++jtemp);
5532             (void) printf("Group = %d, mib_id = %d, "
5533                 "length = %d, valp = 0x%p\n",
5534                 item->group, item->mib_id,
5535                 item->length, item->valp);
5536         }

5538         if (item->group == MIB2_DCCP) {
5539             for (dp = (mib2_dccpConnEntry_t *)item->valp;
5540                  (char *)dp < (char *)item->valp + item->length;
5541                  dp = (mib2_dccpConnEntry_t *)((char *)dp +
```

```

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

5550 (void) fflush(stdout);

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

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

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

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

5583     print_transport_label(attr);

5585     return (B_FALSE);
5586 }

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

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

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

```

```

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

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

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

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

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

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

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

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

5666         switch (item->mib_id) {

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

```

```

5674         "empty");
5675         break;
5676     }

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

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

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

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

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

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

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

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

```

```

5740         mfccp->mfcc_parent);

5742         for (vifi = 0; vifi < MAXVIFS; ++vifi) {
5743             if (mfccp->mfcc_ttls[vifi]) {
5744                 (void) printf(" %u (%u)",
5745                                vifi,
5746                                mfccp->mfcc_ttls[vifi]);
5747             }
5748         }
5749         (void) putchar('\n');
5750     }
5751     (void) printf("\nTotal no. of entries in cache: %d\n",
5752                  nmfc);
5753     break;
5754 }
5755 } /* 'for' loop 1 ends */
5756 (void) putchar('\n');
5757 (void) fflush(stdout);
5758 }

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

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

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

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

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

```

```

5807         if (strcmp(ksp->ks_name, name, len) != 0)
5808             continue; /* 'for' loop 1 */
5810         (void) safe_kstat_read(kc, ksp, NULL);
5812         alloc          = kstat_named_value(ksp, "alloc");
5813         alloc_fail     = kstat_named_value(ksp, "alloc_fail");
5814         buf_size       = kstat_named_value(ksp, "buf_size");
5815         buf_avail      = kstat_named_value(ksp, "buf_avail");
5816         buf_total      = kstat_named_value(ksp, "buf_total");
5817         buf_max        = kstat_named_value(ksp, "buf_max");
5818         buf_inuse      = buf_total - buf_avail;
5820         if (Vflag && prefix) {
5821             (void) snprintf(buf, sizeof (buf), "%s", title,
5822                 ksp->ks_name + len);
5823             (void) printf(" %-18s %6u %9u %11u %11u\n",
5824                 buf, buf_inuse, buf_max, alloc, alloc_fail);
5825         }
5827         total_alloc      += alloc;
5828         total_alloc_fail += alloc_fail;
5829         total_buf_max    += buf_max;
5830         total_buf_inuse  += buf_inuse;
5831         *total_bytes     += (int64_t)buf_inuse * buf_size;
5832     } /* 'for' loop 1 ends */
5834     if (buf_size == 0) {
5835         (void) printf("%-22s [couldn't find statistics for %s]\n",
5836             title, name);
5837         return;
5838     }
5840     if (Vflag && prefix)
5841         (void) snprintf(buf, sizeof (buf), "%s_total", title);
5842     else
5843         (void) snprintf(buf, sizeof (buf), "%s", title);
5845     (void) printf("%-22s %6d %9d %11lld %11d\n", buf,
5846         total_buf_inuse, total_buf_max, total_alloc, total_alloc_fail);
5847 }
5849 static void
5850 m_report(void)
5851 {
5852     int64_t total_bytes = 0;
5854     (void) puts("streams allocation:");
5855     (void) printf("%63s\n", "cumulative allocation");
5856     (void) printf("%63s\n", "current maximum total failures");
5859     kmem_cache_stats("streams",
5860         "stream_head_cache", 0, &total_bytes);
5861     kmem_cache_stats("queues", "queue_cache", 0, &total_bytes);
5862     kmem_cache_stats("mblk", "streams_mblk", 0, &total_bytes);
5863     kmem_cache_stats("dblk", "streams_dblk", 1, &total_bytes);
5864     kmem_cache_stats("linkblk", "linkinfo_cache", 0, &total_bytes);
5865     kmem_cache_stats("syncq", "syncq_cache", 0, &total_bytes);
5866     kmem_cache_stats("qband", "qband_cache", 0, &total_bytes);
5868     (void) printf("\n%lld Kbytes allocated for streams data\n",
5869         total_bytes / 1024);
5871     (void) putchar('\n');

```

```

5872         (void) fflush(stdout);
5873     }
5875     /* ----- */
5877     /*
5878     * Print an IPv4 address. Remove the matching part of the domain name
5879     * from the returned name.
5880     */
5881     static char *
5882     pr_addr(uint_t addr, char *dst, uint_t dstlen)
5883     {
5884         char
5885         struct hostent
5886         static char
5887         static boolean_t
5888         int
5889         if (first) {
5890             first = B_FALSE;
5891             if (sysinfo(SI_HOSTNAME, domain, MAXHOSTNAMELEN) != -1 &&
5892                 (cp = strchr(domain, '.'))) {
5893                 (void) strncpy(domain, cp + 1, sizeof (domain));
5894             } else
5895                 domain[0] = 0;
5896         }
5897         cp = NULL;
5898         if (!Nflag) {
5899             hp = getipnodebyaddr((char *)&addr, sizeof (uint_t), AF_INET,
5900                 &error_num);
5901             if (hp) {
5902                 if ((cp = strchr(hp->h_name, '.')) != NULL &&
5903                     strcasecmp(cp + 1, domain) == 0)
5904                     *cp = 0;
5905                 cp = hp->h_name;
5906             }
5907         }
5908         if (cp != NULL) {
5909             (void) strncpy(dst, cp, dstlen);
5910             dst[dstlen - 1] = 0;
5911         } else {
5912             (void) inet_ntop(AF_INET, (char *)&addr, dst, dstlen);
5913         }
5914         if (hp != NULL)
5915             freehostent(hp);
5916         return (dst);
5917     }
5918 }
5920     /*
5921     * Print a non-zero IPv4 address. Print " --" if the address is zero.
5922     */
5923     static char *
5924     pr_addrnz(ipaddr_t addr, char *dst, uint_t dstlen)
5925     {
5926         if (addr == INADDR_ANY) {
5927             (void) strcpy(dst, " --", dstlen);
5928             return (dst);
5929         }
5930         return (pr_addr(addr, dst, dstlen));
5931     }
5933     /*
5934     * Print an IPv6 address. Remove the matching part of the domain name
5935     * from the returned name.
5936     */
5937     static char *

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

```

```

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

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

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

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

```

```

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

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

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

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

```

```

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

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

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

6242 #define MAX_STRING_SIZE 256

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

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

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

```

```

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

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

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

6286     return (sbuf);
6287 }

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

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

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

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

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

```



```

6334     uint64_t value;

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

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

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

6357     return (value);
6358 }

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

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

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

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

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

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

```

```

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

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

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

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

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

6465     case FK_DST:

```

```

6466     V4MASK_TO_V6(IP_HOST_MASK, newf->u.a.f_mask);
6467     if (strcmp(arg, "any") == 0) {
6468         /* Special semantics; any address *but* zero */
6469         newf->u.a.f_address = NULL;
6470         (void) memset(&newf->u.a.f_mask, 0,
6471             sizeof (newf->u.a.f_mask));
6472         break;
6473     }
6474     if (strcmp(arg, "none") == 0) {
6475         newf->u.a.f_address = NULL;
6476         break;
6477     }
6478     if ((cp = strrchr(arg, '/')) != NULL)
6479         *cp++ = '\0';
6480     hp = getipnodebyname(arg, AF_INET6, AI_V4MAPPED|AI_ALL,
6481         &error_num);
6482     if (hp == NULL)
6483         fatal(1, "%s: invalid or unknown host address\n", arg);
6484     newf->u.a.f_address = hp;
6485     if (cp == NULL) {
6486         V4MASK_TO_V6(IP_HOST_MASK, newf->u.a.f_mask);
6487     } else {
6488         val = strtoul(cp, &cp2, 0);
6489         if (cp != cp2 && cp2[0] == '\0') {
6490             /*
6491              * If decode as "/" works, then translate
6492              * into a mask.
6493              */
6494             if (hp->h_addr_list[0] != NULL &&
6495                 /* LINTED: (note 1) */
6496                 IN6_IS_ADDR_V4MAPPED((in6_addr_t *)
6497                     hp->h_addr_list[0])) {
6498                 maxv = IP_ABITS;
6499             } else {
6500                 maxv = IPV6_ABITS;
6501             }
6502             if (val < 0 || val >= maxv)
6503                 fatal(1, "%d: not in range 0 to %d\n",
6504                     val, maxv - 1);
6505             if (maxv == IP_ABITS)
6506                 val += IPV6_ABITS - IP_ABITS;
6507             ucp = newf->u.a.f_mask.s6_addr;
6508             while (val >= 8)
6509                 *ucp++ = 0xff, val -= 8;
6510             *ucp++ = (0xff << (8 - val)) & 0xff;
6511             while (ucp < newf->u.a.f_mask.s6_addr +
6512                 sizeof (newf->u.a.f_mask.s6_addr))
6513                 *ucp++ = 0;
6514             /* Otherwise, try as numeric address */
6515         } else if (inet_pton(AF_INET6,
6516             cp, &newf->u.a.f_mask) <= 0) {
6517             fatal(1, "%s: illegal mask format\n", cp);
6518         }
6519     }
6520     break;
6521
6522 case FK_FLAGS:
6523     if (*arg == '+') {
6524         newf->u.f.f_flagset = flag_bits(arg + 1);
6525         newf->u.f.f_flagclear = 0;
6526     } else if (*arg == '-') {
6527         newf->u.f.f_flagset = 0;
6528         newf->u.f.f_flagclear = flag_bits(arg + 1);
6529     } else {
6530         newf->u.f.f_flagset = flag_bits(arg);
6531         newf->u.f.f_flagclear = ~newf->u.f.f_flagset;

```

```

6532     }
6533     break;
6534
6535     default:
6536         assert(0);
6537     }
6538     newf->f_next = filters[idx];
6539     filters[idx] = newf;
6540 }
6541
6542 /* Determine if user wants this address family printed. */
6543 static boolean_t
6544 family_selected(int family)
6545 {
6546     const filter_t *fp;
6547
6548     if (v4compat && family == AF_INET6)
6549         return (B_FALSE);
6550     if ((fp = filters[FK_AF]) == NULL)
6551         return (B_TRUE);
6552     while (fp != NULL) {
6553         if (fp->u.f_family == family)
6554             return (B_TRUE);
6555         fp = fp->f_next;
6556     }
6557     return (B_FALSE);
6558 }
6559
6560 /*
6561  * Convert the interface index to a string using the buffer 'ifname', which
6562  * must be at least LIFNAMSIZ bytes. We first try to map it to name. If that
6563  * fails (e.g., because we're inside a zone and it does not have access to
6564  * interface for the index in question), just return "if#<num>".
6565  */
6566 static char *
6567 ifindex2str(uint_t ifindex, char *ifname)
6568 {
6569     if (if_indextoname(ifindex, ifname) == NULL)
6570         (void) snprintf(ifname, LIFNAMSIZ, "if#%d", ifindex);
6571
6572     return (ifname);
6573 }
6574
6575 /*
6576  * print the usage line
6577  */
6578 static void
6579 usage(char *cmdname)
6580 {
6581     (void) fprintf(stderr, "usage: %s [-anv] [-f address_family] "
6582         "[-T d|u]\n", cmdname);
6583     (void) fprintf(stderr, "        %s [-n] [-f address_family] "
6584         "[-P protocol] [-T d|u] [-g | -p | -s [interval [count]]]\n",
6585         cmdname);
6586     (void) fprintf(stderr, "        %s -m [-v] [-T d|u] "
6587         "[-interval [count]]\n", cmdname);
6588     (void) fprintf(stderr, "        %s -i [-I interface] [-an] "
6589         "[-f address_family] [-T d|u] [-interval [count]]\n", cmdname);
6590     (void) fprintf(stderr, "        %s -r [-anv] "
6591         "[-f address_family|filter] [-T d|u]\n", cmdname);
6592     (void) fprintf(stderr, "        %s -M [-ns] [-f address_family] "
6593         "[-T d|u]\n", cmdname);
6594     (void) fprintf(stderr, "        %s -D [-I interface] "
6595         "[-f address_family] [-T d|u]\n", cmdname);
6596     exit(EXIT_FAILURE);
6597 }

```

```
6599 /*
6600  * fatal: print error message to stderr and
6601  * call exit(errcode)
6602  */
6603 /*PRINTFLIKE2*/
6604 static void
6605 fatal(int errcode, char *format, ...)
6606 {
6607     va_list argp;
6608
6609     if (format == NULL)
6610         return;
6611
6612     va_start(argp, format);
6613     (void) vfprintf(stderr, format, argp);
6614     va_end(argp);
6615
6616     exit(errcode);
6617 }
```

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

1

57122 Sat Aug 18 10:36:56 2012

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

dccp: properties

_____unchanged_portion_omitted_____

```
634 /*
635  * Properties to be displayed is in 'statep->sps_proplist'. If it is NULL,
636  * for all the properties for the specified object, relevant information, will
637  * be displayed. Otherwise, for the selected property set, display relevant
638  * information
639  */
640 static void
641 show_properties(void *arg, int prop_class)
642 {
643     show_prop_state_t      *statep = arg;
644     nvlist_t               *nvl = statep->sps_proplist;
645     uint_t                 proto = statep->sps_proto;
646     nvpair_t               *curr_nvp;
647     char                   *buf, *name;
648     ipadm_status_t         status;
649
650     /* allocate sufficient buffer to hold a property value */
651     if ((buf = malloc(MAXPROPVLEN)) == NULL)
652         die("insufficient memory");
653     statep->sps_propval = buf;
654
655     /* if no properties were specified, display all the properties */
656     if (nvl == NULL) {
657         (void) ipadm_walk_proptbl(proto, prop_class, show_property,
658             statep);
659     } else {
660         for (curr_nvp = nvlist_next_nvpair(nvl, NULL); curr_nvp;
661             curr_nvp = nvlist_next_nvpair(nvl, curr_nvp)) {
662             name = nvpair_name(curr_nvp);
663             status = ipadm_walk_prop(name, proto, prop_class,
664                 show_property, statep);
665             if (status == IPADM_PROP_UNKNOWN)
666                 (void) show_property(statep, name, proto);
667         }
668     }
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;
```

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

2

```
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/cmd-inet/usr.sbin/snoop/Makefile

1

2254 Sat Aug 18 10:36:57 2012

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/Makefile

dccp: snoop, build system fixes

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

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

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

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

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

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

53 .KEEP_STATE:

55 .PARALLEL: $(OBJS)

57 all:      $(PROG)

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

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/Makefile

2

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

63 install:      all $(ROOTUSRSBINPROG)

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

68 lint:      lint_SRCS

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

```

*****
12779 Sat Aug 18 10:36:57 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop.h
dccp: snoop, build system fixes
*****
_____unchanged_portion_omitted_____

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

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

```

```

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

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

```

```

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

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

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

```

```

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

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

318 extern int quitting;
319 extern boolean_t Iflag, Pflag, rflag;

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

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

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

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

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

354 #ifdef __cplusplus
355 }
356 #endif

358 #endif /* _SNOOP_H */

```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_dccp.c

1

```
*****
10243 Sat Aug 18 10:36:57 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_dccp.c
dccp: snoop, build system fixes
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, 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 2005 Sun Microsystems, Inc. All rights reserved.
24  * Copyright 2012 David Hoepfner. All rights reserved.
25  */

27 #include <stdio.h>
28 #include <ctype.h>
29 #include <string.h>
30 #include <fcntl.h>
31 #include <string.h>
32 #include <sys/types.h>
33 #include <sys/time.h>

35 #include <sys/socket.h>
36 #include <sys/sockio.h>
37 #include <net/if.h>
38 #include <netinet/in_systm.h>
39 #include <netinet/in.h>
40 #include <netinet/ip.h>
41 #include <netinet/if_ether.h>
42 #include <netinet/dccp.h>
43 #include "snoop.h"

45 /*
46  * Snoop interpreter for DCCP (RFC4340)
47  *
48  */

50 extern char *dlc_header;

52 static char *get_type(uint8_t);
53 static void print_dccpoptions_summary(uchar_t *, uchar_t *);
54 static void print_dccpoptions(uchar_t *, uchar_t *);

56 static char *
57 get_type(uint8_t type)
58 {
59     switch (type) {
60     case 0:
61         return ("DCCP-Request");
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_dccp.c

2

```
62     case 1:
63         return ("DCCP-Response");
64     case 2:
65         return ("DCCP-Data");
66     case 3:
67         return ("DCCP-Ack");
68     case 4:
69         return ("DCCP-DataAck");
70     case 5:
71         return ("DCCP-CloseReq");
72     case 6:
73         return ("DCCP-Close");
74     case 7:
75         return ("DCCP-Reset");
76     case 8:
77         return ("DCCP-Sync");
78     case 9:
79         return ("DCCP-SyncAck");
80     case 10 ... 15:
81         return ("Reserved");

83     default:
84         return ("Unknown");
85     }
86 }

88 int
89 interpret_dccp(int flags, struct dccphdr *dccp, int iplen, int fraglen)
90 {
91     char *data;
92     char *line;
93     char *newline;
94     uint64_t seq;
95     uint64_t ack;
96     int hdrlen;
97     int dccplen;
98     int option_offset;
99     int i;

101     hdrlen = dccp->dh_offset * 4;
102     data = (char *)dccp + hdrlen;
103     dccplen = iplen - hdrlen;
104     fraglen -= hdrlen;

106     if (fraglen < 0) {
107         return (fraglen + hdrlen); /* Incomplete header */
108     }

110     if (fraglen > dccplen) {
111         fraglen = dccplen;
112     }

114     if (dccp->dh_x == 1) {
115         switch (dccp->dh_type) {
116         case 0: /* DCCP-Request */
117             option_offset = 20;
118             break;
119         case 1: /* DCCP-Response */
120             option_offset = 28;
121             break;
122         case 7: /* DCCP-Reset */
123             option_offset = 28;
124             break;
125         case 8: /* DCCP-Sync */
126         case 9: /* DCCP-SyncAck */
127             option_offset = 24;
```



```

128             break;
130         default:
131             option_offset = 20;
132             break;
133     }

135     /* Sequence number */
136     seq = ntohs(dccp->dh_seq);
137     seq <= 32;
138     seq += (uint32_t)dccp + sizeof (struct dccphdr);

140     ack = (uint32_t)(dccp + 16) & 0xffff;
141 } else {
142     switch (dccp->dh_type) {
143     case 0: /* DCCP-Request */
144     case 1: /* DCCP-Response */
145     case 7: /* DCCP-Reset */
146     case 8: /* DCCP-Sync */
147     case 9: /* DCCP-SyncAck */
148         break;

150     default:
151         option_offset = 20;
152         break;
153     }

155     /* Sequence number */
156     seq = dccp->dh_res_seq < 16;
157     seq |= ntohs(dccp->dh_seq);
158 }

160 if (flags & F_SUM) {
161     line = get_sum_line();
162     endl = line + MAXLINE;

164     (void) snprintf(line, endl - line, "DCCP D=%d S=%d",
165         ntohs(dccp->dh_dport), ntohs(dccp->dh_sport));
166     line += strlen(line);

168     (void) snprintf(line, endl - line, "CCval=%d CsCov=%d",
169         dccp->dh_ccval, dccp->dh_cscov);

171     (void) snprintf(line, endl - line, " Seq=%u Len=%d",
172         seq, dccplen);
173     line += strlen(line);

175     /*
176      * All packets except DCCP-Request and DCCP-Data carry
177      * an acknowledgement number.
178      */
179     if (dccp->dh_type != 0 && dccp->dh_type != 2) {
180         (void) snprintf(line, endl - line, " Ack=%u",
181             seq, dccplen);
182     }
183     line += strlen(line);

185     print_dccpoptions_summary((uchar_t *)dccp + option_offset,
186         (uchar_t *)dccp + dccp->dh_offset * 4);
187 }

190 if (flags & F_DTAIL) {
191     show_header("DCCP: ", "DCCP Header", dccplen);
192     show_space();

```

```

194     (void) sprintf(get_line((char *) (uintptr_t) dccp->dh_sport -
195         dlc_header, 2), "Source port = %d",
196         ntohs(dccp->dh_sport));

198     (void) sprintf(get_line((char *) (uintptr_t) dccp->dh_dport -
199         dlc_header, 2), "Destination port = %d",
200         ntohs(dccp->dh_dport));

202     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_offset -
203         dlc_header) + 4, 1), "Data offset = %d bytes",
204         dccp->dh_offset * 4);

206     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_ccval -
207         dlc_header) + 4, 1), "CCval = %d",
208         dccp->dh_ccval);

210     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_cscov -
211         dlc_header) + 4, 1), "Checksum coverage (CsCov) = %d",
212         dccp->dh_cscov);

214     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_sum -
215         dlc_header) + 4, 1), "Checksum = 0x%04x",
216         ntohs(dccp->dh_sum));

218     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_type -
219         dlc_header) + 4, 1), "Type = %d (%s)",
220         dccp->dh_type, get_type(dccp->dh_type));

222     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_x -
223         dlc_header) + 4, 1), "Extended sequence numbers = %d",
224         dccp->dh_x);

226     (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_seq -
227         dlc_header) + 4, 1), "Sequence number = %d",
228         seq);

230     /*
231      * All packets except DCCP-Request and DCCP-Data carry
232      * an acknowledgement number.
233      */
234     if (dccp->dh_type != 0 && dccp->dh_type != 2) {
235         (void) sprintf(get_line(((char *) (uintptr_t) dccp->dh_seq -
236         dlc_header) + 4, 1), "Acknowledgement number = %d",
237         seq);
238     }

240     print_dccpoptions((uchar_t *)dccp + option_offset,
241         (uchar_t *)dccp + dccp->dh_offset * 4);

243     show_space();
244 }

246     return (dccplen);
247 }

249 static void
250 print_dccpoptions_summary(uchar_t *up, uchar_t *end)
251 {
252     uchar_t      *value;
253     uint8_t      option_type;
254     uint8_t      option_length;
255     int          len;
256     boolean_t    mandatory = B_FALSE;

258     while (up != end) {
259         option_length = 0;

```

```

260         option_type = *up++;
261
262         /*
263          * Is this a variable length option?
264          */
265         if (option_type > 31) {
266             option_length = *up++;
267             option_length -= 2;
268             value = up;
269
270             up += option_length;
271         }
272         switch (option_type) {
273             case 0: /* Padding */
274                 break;
275             case 1: /* Mandatory */
276                 mandatory = B_TRUE;
277                 break;
278             case 2: /* Slow receiver */
279                 break;
280             case 3 ... 31: /* Reserved */
281                 break;
282             case 32: /* Change L */
283                 break;
284             case 33: /* Confirm L */
285                 break;
286             case 34: /* Change R */
287                 break;
288             case 35: /* Confirm R */
289                 break;
290             case 36: /* Init cookie */
291                 break;
292             case 37: /* NDP count */
293                 break;
294             case 38: /* Ack Vector 0 */
295                 break;
296             case 39: /* Ack vector 1 */
297                 break;
298             case 40: /* Data dropped */
299                 break;
300             case 41: /* Timestamp */
301                 break;
302             case 42: /* Timestamp echo */
303                 break;
304             case 43: /* Elapsed time */
305                 break;
306             case 44: /* Data checksum */
307                 break;
308
309             default:
310                 break;
311         }
312
313         if (option_type != 1) {
314             mandatory = B_FALSE;
315         }
316     }
317 }
318
319 static void
320 print_dccpoptions(uchar_t *up, uchar_t *end)
321 {
322     uchar_t    *value;
323     uint8_t    option_type;
324     uint8_t    option_length;
325     int        len;

```

```

326     boolean_t    mandatory = B_FALSE;
327
328     if (up == end) {
329         (void) sprintf(get_line((char *)&up - dlc_header, 1),
330             "No options");
331         return;
332     }
333
334     (void) sprintf(get_line((char *)&up - dlc_header, 1),
335         "Options: (%d bytes)", (int)(end - up));
336
337     while (up != end) {
338         option_length = 0;
339         option_type = *up++;
340
341         /*
342          * Is this a variable length option?
343          */
344         if (option_type > 31) {
345             option_length = *up++;
346             option_length -= 2;
347             value = up;
348
349             up += option_length;
350         }
351
352         switch (option_type) {
353             case 0: /* Padding */
354                 (void) sprintf(get_line(((char *) (uintptr_t)up -
355                     dlc_header) + 4, 1), "Padding");
356                 break;
357             case 1: /* Mandatory */
358                 (void) sprintf(get_line(((char *) (uintptr_t)up -
359                     dlc_header) + 4, 1), "Mandatory");
360                 mandatory = B_TRUE;
361                 break;
362             case 2: /* Slow receiver */
363                 (void) sprintf(get_line(((char *) (uintptr_t)up -
364                     dlc_header) + 4, 1), "Slow receiver");
365                 break;
366             case 3 ... 31: /* Reserved */
367                 (void) sprintf(get_line(((char *) (uintptr_t)up -
368                     dlc_header) + 4, 1), "Reserved");
369                 break;
370             case 32: /* Change L */
371                 (void) sprintf(get_line(((char *) (uintptr_t)up -
372                     dlc_header) + 4, 1), "Change L");
373                 break;
374             case 33: /* Confirm L */
375                 (void) sprintf(get_line(((char *) (uintptr_t)up -
376                     dlc_header) + 4, 1), "Confirm L");
377                 break;
378             case 34: /* Change R */
379                 (void) sprintf(get_line(((char *) (uintptr_t)up -
380                     dlc_header) + 4, 1), "Change R");
381                 break;
382             case 35: /* Confirm R */
383                 (void) sprintf(get_line(((char *) (uintptr_t)up -
384                     dlc_header) + 4, 1), "Confirm R");
385                 break;
386             case 36: /* Init cookie */
387                 (void) sprintf(get_line(((char *) (uintptr_t)up -
388                     dlc_header) + 4, 1), "Init cookie");
389                 break;
390             case 37: /* NDP count */
391                 (void) sprintf(get_line(((char *) (uintptr_t)up -

```

```
392         dlc_header) + 4, 1), "NDP count");
393     break;
394     case 38: /* Ack vector 0 */
395         (void) sprintf(get_line(((char *) (uintptr_t) up -
396         dlc_header) + 4, 1), "Ack vector 0");
397     break;
398     case 39: /* Ack vector 1 */
399         (void) sprintf(get_line(((char *) (uintptr_t) up -
400         dlc_header) + 4, 1), "Ack vector 1");
401     break;
402     case 40: /* Data dropped */
403         (void) sprintf(get_line(((char *) (uintptr_t) up -
404         dlc_header) + 4, 1), "Data dropped");
405     break;
406     case 41: /* Timestamp */
407         (void) sprintf(get_line(((char *) (uintptr_t) up -
408         dlc_header) + 4, 1), "Timestamp");
409     break;
410     case 42: /* Timestamp echo */
411         (void) sprintf(get_line(((char *) (uintptr_t) up -
412         dlc_header) + 4, 1), "Timestamp echo");
413     break;
414     case 43: /* Elapsed time */
415         (void) sprintf(get_line(((char *) (uintptr_t) up -
416         dlc_header) + 4, 1), "Elapsed time");
417     break;
418     case 44: /* Data checksum */
419         (void) sprintf(get_line(((char *) (uintptr_t) up -
420         dlc_header) + 4, 1), "Data checksum");
421     break;
422
423     default:
424         (void) sprintf(get_line(((char *) (uintptr_t) up -
425         dlc_header) + 4, 1), "Unknown");
426     break;
427 }
428
429 if (option_type != 1) {
430     mandatory = B_FALSE;
431 }
432 }
433 }
434 #endif /* ! codereview */
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_filter.c

1

63156 Sat Aug 18 10:36:58 2012

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_filter.c

dccp: options and features

_____unchanged_portion_omitted_____

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

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

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_filter.c

2

```
1381     0,            0, 0, 0, 0,            0,      0
1382 };
1384 static match_type_t iptun_match_types[] = {
1385     "ip",          0, 1, IPPROTO_ENCAP,    -1,    OP_OFFSET_ETHERTYPE,
1386     "ip6",         0, 1, IPPROTO_IPV6,    -1,    OP_OFFSET_ETHERTYPE,
1387     "tcp",         9, 1, IPPROTO_TCP,      0,      OP_OFFSET_LINK,
1388     "tcp",         6, 1, IPPROTO_TCP,      1,      OP_OFFSET_LINK,
1389     "udp",         9, 1, IPPROTO_UDP,      0,      OP_OFFSET_LINK,
1390     "udp",         6, 1, IPPROTO_UDP,      1,      OP_OFFSET_LINK,
1391     "icmp",        9, 1, IPPROTO_ICMP,     0,      OP_OFFSET_LINK,
1392     "icmp6",       6, 1, IPPROTO_ICMPV6,   1,      OP_OFFSET_LINK,
1393     "ospf",        9, 1, IPPROTO_OSPF,     0,      OP_OFFSET_LINK,
1394     "ospf",        6, 1, IPPROTO_OSPF,     1,      OP_OFFSET_LINK,
1395     "ip-in-ip",    9, 1, IPPROTO_ENCAP,    0,      OP_OFFSET_LINK,
1396     "esp",         9, 1, IPPROTO_ESP,      0,      OP_OFFSET_LINK,
1397     "esp",         6, 1, IPPROTO_ESP,      1,      OP_OFFSET_LINK,
1398     "ah",          9, 1, IPPROTO_AH,       0,      OP_OFFSET_LINK,
1399     "ah",          6, 1, IPPROTO_AH,       1,      OP_OFFSET_LINK,
1400     "sctp",        9, 1, IPPROTO_SCTP,     0,      OP_OFFSET_LINK,
1401     "sctp",        6, 1, IPPROTO_SCTP,     1,      OP_OFFSET_LINK,
1402     "dccp",        9, 1, IPPROTO_DCCP,     0,      OP_OFFSET_LINK,
1403     "dccp",        6, 1, IPPROTO_DCCP,     1,      OP_OFFSET_LINK,
1404 #endif /* ! codereview */
1405     0,            0, 0, 0, 0,            0,      0
1406 };
```

```
1408 static void
1409 generate_check(match_type_t match_types[], int index, int type)
1410 {
1411     match_type_t *mtp = &match_types[index];
1412     /*
1413      * Note: this code assumes the above dependencies are
1414      * not cyclic. This *should* always be true.
1415      */
1416     if (mtp->m_depend != -1)
1417         generate_check(match_types, mtp->m_depend, type);
1419     emitop(mtp->m_optype);
1420     load_value(mtp->m_offset, mtp->m_size);
1421     load_const(mtp->m_value);
1422     emitop(OP_OFFSET_POP);
1424     emitop(OP_EQ);
1426     if (mtp->m_depend != -1)
1427         emitop(OP_AND);
1428 }
```

```
1430 /*
1431  * Generate code based on the keyword argument.
1432  * This word is looked up in the match_types table
1433  * and checks a field within the packet for a given
1434  * value e.g. ether or ip type field. The match
1435  * can also have a dependency on another entry e.g.
1436  * "tcp" requires that the packet also be "ip".
1437  */
1438 static int
1439 comparison(char *s)
1440 {
1441     unsigned int    i, n_checks = 0;
1442     match_type_t    *match_types;
1444     switch (interface->mac_type) {
1445     case DL_ETHER:
1446         match_types = ether_match_types;
```

```

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

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

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

1470 return (n_checks > 0);
1471 }

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

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

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

1507     found_host = 0;

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

```

```

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

```

```

1579         break;
1580     default:
1581         found_host = 0;
1582     }
1583
1584     if (!found_host) {
1585         if (error_num == TRY_AGAIN) {
1586             pr_err("could not resolve %s (try again later)",
1587                 hostname);
1588         } else {
1589             pr_err("could not resolve %s", hostname);
1590         }
1591     }
1592 }
1593
1594 switch (which) {
1595 case TO:
1596     addr4offset = IPV4_DSTADDR_OFFSET;
1597     addr6offset = IPV6_DSTADDR_OFFSET;
1598     break;
1599 case FROM:
1600     addr4offset = IPV4_SRCADDR_OFFSET;
1601     addr6offset = IPV6_SRCADDR_OFFSET;
1602     break;
1603 case ANY:
1604     addr4offset = -1;
1605     addr6offset = -1;
1606     break;
1607 }
1608
1609 /*
1610  * The code below generates the filter.
1611  */
1612 if (hp != NULL && hp->h_addrtype == AF_INET) {
1613     ethertype_match(interface->network_type_ip);
1614     emitop(OP_BRFL);
1615     n = chain(n);
1616     emitop(OP_OFFSET_LINK);
1617     h_addr_index = 0;
1618     addr4ptr = (uint_t *)hp->h_addr_list[h_addr_index];
1619     while (addr4ptr != NULL) {
1620         if (addr4offset == -1) {
1621             compare_addr_v4(IPV4_SRCADDR_OFFSET, 4,
1622                 *addr4ptr);
1623             emitop(OP_BRTR);
1624             m = chain(m);
1625             compare_addr_v4(IPV4_DSTADDR_OFFSET, 4,
1626                 *addr4ptr);
1627         } else {
1628             compare_addr_v4(addr4offset, 4, *addr4ptr);
1629         }
1630         addr4ptr = (uint_t *)hp->h_addr_list[++h_addr_index];
1631         if (addr4ptr != NULL) {
1632             emitop(OP_BRTR);
1633             m = chain(m);
1634         }
1635     }
1636     if (m != 0) {
1637         resolve_chain(m);
1638     }
1639     emitop(OP_OFFSET_POP);
1640     resolve_chain(n);
1641 } else {
1642     /* first pass: IPv4 addresses */
1643     h_addr_index = 0;
1644     addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];

```

```

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

```

```

1711         } else {
1712             compare_addr_v6(addr6offset, 16,
1713                             *addr6ptr);
1714         }
1715     }
1716     addr6ptr = (struct in6_addr *)
1717         hp->h_addr_list[++h_addr_index];
1718 }
1719 if (m != 0) {
1720     resolve_chain(m);
1721 }
1722 emitop(OP_OFFSET_POP);
1723 resolve_chain(n);
1724 }
1725
1726 /* only free struct hostent returned by getipnodebyname() */
1727 if (freehp) {
1728     freehostent(hp);
1729 }
1730 }
1731
1732 /*
1733  * Match on zoneid. The arg zone passed in is in network byte order.
1734  */
1735 static void
1736 zone_match(enum direction which, uint32_t zone)
1737 {
1738     switch (which) {
1739     case TO:
1740         compare_value_zone(IPNET_DSTZONE_OFFSET, zone);
1741         break;
1742     case FROM:
1743         compare_value_zone(IPNET_SRCZONE_OFFSET, zone);
1744         break;
1745     case ANY:
1746         compare_value_zone(IPNET_SRCZONE_OFFSET, zone);
1747         compare_value_zone(IPNET_DSTZONE_OFFSET, zone);
1748         emitop(OP_OR);
1749     }
1750 }
1751 }
1752
1753 /*
1754  * Generate code to match an AppleTalk address. The address
1755  * must be given as two numbers with a dot between
1756  */
1757 static void
1758 ataddr_match(enum direction which, char *hostname)
1759 {
1760     uint_t net;
1761     uint_t node;
1762     uint_t m, n;
1763
1764     sscanf(hostname, "%u.%u", &net, &node);
1765
1766     emitop(OP_OFFSET_LINK);
1767     switch (which) {
1768     case TO:
1769         compare_value(AT_DST_NET_OFFSET, 2, net);
1770         emitop(OP_BRFL);
1771         m = chain(0);
1772         compare_value(AT_DST_NODE_OFFSET, 1, node);
1773         resolve_chain(m);
1774         break;
1775     case FROM:

```

```

1777         compare_value(AT_SRC_NET_OFFSET, 2, net);
1778         emitop(OP_BRFL);
1779         m = chain(0);
1780         compare_value(AT_SRC_NODE_OFFSET, 1, node);
1781         resolve_chain(m);
1782         break;
1783     case ANY:
1784         compare_value(AT_DST_NET_OFFSET, 2, net);
1785         emitop(OP_BRFL);
1786         m = chain(0);
1787         compare_value(AT_DST_NODE_OFFSET, 1, node);
1788         resolve_chain(m);
1789         emitop(OP_BRTR);
1790         n = chain(0);
1791         compare_value(AT_SRC_NET_OFFSET, 2, net);
1792         emitop(OP_BRFL);
1793         m = chain(0);
1794         compare_value(AT_SRC_NODE_OFFSET, 1, node);
1795         resolve_chain(m);
1796         resolve_chain(n);
1797         break;
1798     }
1799     emitop(OP_OFFSET_POP);
1800 }
1801
1802 /*
1803  * Compare ethernet addresses. The address may
1804  * be provided either as a hostname or as a
1805  * 6 octet colon-separated address.
1806  */
1807 static void
1808 etheraddr_match(enum direction which, char *hostname)
1809 {
1810     uint_t addr;
1811     ushort_t *addrp;
1812     int to_offset, from_offset;
1813     struct ether_addr e, *ep = NULL;
1814     int m;
1815
1816     /*
1817      * First, check the interface type for whether src/dest address
1818      * is determinable; if not, retreat early.
1819      */
1820     switch (interface->mac_type) {
1821     case DL_ETHER:
1822         from_offset = ETHERADDRL;
1823         to_offset = 0;
1824         break;
1825
1826     case DL_IB:
1827         /*
1828          * If an ethernet address is attempted to be used
1829          * on an IPoIB interface, flag error. Link address
1830          * based filtering is unsupported on IPoIB, so there
1831          * is no ipibaddr_match() or parsing support for IPoIB
1832          * 20 byte link addresses.
1833          */
1834         pr_err("filter option unsupported on media");
1835         break;
1836
1837     case DL_FDDI:
1838         from_offset = 7;
1839         to_offset = 1;
1840         break;
1841
1842     default:

```

```

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

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

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

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

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

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

```

```

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

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

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

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

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

```



```

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

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

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

2006     emitop(OP_OFFSET_IP);

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

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

2040     if (isdigit(*progname)) {

```

```

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

```

```

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

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

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

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

```

```

2371         break;
2372     }

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

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

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

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

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

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

2436     if (EQ("greater")) {

```

```

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

```

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

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

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

2662     return (0);
2663 }

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

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

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

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

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

```

```

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

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

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

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

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

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

```

```

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

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

2778      p = str;

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

2791      return (str);
2792  }

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

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

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

```

```

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

```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ip.c

1

```
*****
39402 Sat Aug 18 10:36:58 2012
new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ip.c
dccp: snoop, build system fixes
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #include <stdio.h>
27 #include <string.h>
28 #include <fcntl.h>
29 #include <string.h>
30 #include <sys/types.h>
31 #include <sys/time.h>

33 #include <sys/stropts.h>
34 #include <sys/socket.h>
35 #include <net/if.h>
36 #include <netinet/in_systm.h>
37 #include <netinet/in.h>
38 #include <netinet/ip.h>
39 #include <netinet/ip6.h>
40 #include <netinet/ip_icmp.h>
41 #include <netinet/icmp6.h>
42 #include <netinet/if_ether.h>
43 #include <inet/ip.h>
44 #include <inet/ip6.h>
45 #include <arpa/inet.h>
46 #include <netdb.h>
47 #include <tsol/label.h>
48 #include <sys/tsol/tndb.h>
49 #include <sys/tsol/label_macro.h>

51 #include "snoop.h"

54 /*
55 * IPv6 extension header masks. These are used by the print_ipv6_extensions()
56 * function to return information to the caller about which extension headers
57 * were processed. This can be useful if the caller wants to know if the
58 * packet is an IPv6 fragment, for example.
59 */
60 #define SNOOP_HOPOPTS 0x01U
61 #define SNOOP_ROUTING 0x02U
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ip.c

2

```
62 #define SNOOP_DSTOPTS 0x04U
63 #define SNOOP_FRAGMENT 0x08U
64 #define SNOOP_AH 0x10U
65 #define SNOOP_ESP 0x20U
66 #define SNOOP_IPV6 0x40U

68 static void prt_routing_hdr(int, const struct ip6_rthdr *);
69 static void prt_fragment_hdr(int, const struct ip6_frag *);
70 static void prt_hbh_options(int, const struct ip6_hbh *);
71 static void prt_dest_options(int, const struct ip6_dest *);
72 static void print_route(const uchar_t *);
73 static void print_ipoptions(const uchar_t *, int);
74 static void print_ripso(const uchar_t *);
75 static void print_cipso(const uchar_t *);

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

81 int
82 interpret_ip(int flags, const struct ip *ip, int fraglen)
83 {
84     uchar_t *data;
85     char buff[24];
86     boolean_t isfrag = B_FALSE;
87     boolean_t morefrag;
88     uint16_t fragoffset;
89     int hdrlen;
90     uint16_t iplen, uitmp;

92     if (ip->ip_v == IPV6_VERSION) {
93         iplen = interpret_ipv6(flags, (ip6_t *)ip, fraglen);
94         return (iplen);
95     }

97     if (encap_levels == 0)
98         total_encap_levels = 0;
99     encap_levels++;
100    total_encap_levels++;

102    hdrlen = ip->ip_hl * 4;
103    data = ((uchar_t *)ip) + hdrlen;
104    iplen = ntohs(ip->ip_len) - hdrlen;
105    fraglen -= hdrlen;
106    if (fraglen > iplen)
107        fraglen = iplen;
108    if (fraglen < 0) {
109        (void) snprintf(get_sum_line(), MAXLINE,
110            "IP truncated: header missing %d bytes", -fraglen);
111        encap_levels--;
112        return (fraglen + iplen);
113    }
114    /*
115     * We flag this as a fragment if the more fragments bit is set, or
116     * if the fragment offset is non-zero.
117     */
118    morefrag = (ntohs(ip->ip_off) & IP_MF) == 0 ? B_FALSE : B_TRUE;
119    fragoffset = (ntohs(ip->ip_off) & 0x1FFF) * 8;
120    if (morefrag || fragoffset != 0)
121        isfrag = B_TRUE;

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

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



```

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

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

```

```

194         "do not fragment", "may fragment"));
195     (void) snprintf(get_line(0, 0), get_line_remain(), "%s",
196         getflag(uitmp >> 8, IP_MF >> 8,
197         "more fragments", "last fragment"));
198     (void) snprintf(get_line(0, 0), get_line_remain(),
199         "Fragment offset = %u bytes",
200         fragoffset);
201     (void) snprintf(get_line(0, 0), get_line_remain(),
202         "Time to live = %d seconds/hops",
203         ip->ip_ttl);
204     (void) snprintf(get_line(0, 0), get_line_remain(),
205         "Protocol = %d (%s)", ip->ip_p,
206         getproto(ip->ip_p));
207     /*
208      * XXX need to compute checksum and print whether it's correct
209      */
210     (void) snprintf(get_line(0, 0), get_line_remain(),
211         "Header checksum = %04x",
212         ntohs(ip->ip_sum));
213     (void) snprintf(get_line(0, 0), get_line_remain(),
214         "Source address = %s, %s",
215         inet_ntoa(ip->ip_src), addrtoname(AF_INET, &ip->ip_src));
216     (void) snprintf(get_line(0, 0), get_line_remain(),
217         "Destination address = %s, %s",
218         inet_ntoa(ip->ip_dst), addrtoname(AF_INET, &ip->ip_dst));
219
220     /* Print IP options - if any */
221
222     print_ipoptions((const uchar_t *) (ip + 1),
223         hdrhlen - sizeof (struct ip));
224     show_space();
225 }

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

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

```

```

260         case IPPROTO_GGP:
261             break;
262         case IPPROTO_TCP:
263             (void) interpret_tcp(flags,
264                 (struct tcphdr *)data, iplen, fraglen);
265             break;
266
267         case IPPROTO_ESP:
268             (void) interpret_esp(flags, data, iplen,
269                 fraglen);
270             break;
271         case IPPROTO_AH:
272             (void) interpret_ah(flags, data, iplen,
273                 fraglen);
274             break;
275
276         case IPPROTO_OSPF:
277             interpret_ospf(flags, data, iplen, fraglen);
278             break;
279
280         case IPPROTO_EGP:
281         case IPPROTO_PUP:
282             break;
283         case IPPROTO_UDP:
284             (void) interpret_udp(flags,
285                 (struct udphdr *)data, iplen, fraglen);
286             break;
287
288         case IPPROTO_IDP:
289         case IPPROTO_HELLO:
290         case IPPROTO_ND:
291         case IPPROTO_RAW:
292             break;
293         case IPPROTO_IPV6: /* IPV6 encap */
294             /* LINTED: alignment */
295             (void) interpret_ipv6(flags, (ip6_t *)data,
296                 iplen);
297             break;
298         case IPPROTO_SCTP:
299             (void) interpret_sctp(flags,
300                 (struct sctp_hdr *)data, iplen, fraglen);
301             break;
302         case IPPROTO_DCCP:
303             (void) interpret_dccp(flags,
304                 (struct dccphdr *)data, iplen, fraglen);
305             break;
306 #endif /* ! codereview */
307     }
308 }
309
310     encap_levels--;
311     return (iplen);
312 }
313
314 int
315 interpret_ipv6(int flags, const ip6_t *ip6h, int fraglen)
316 {
317     uint8_t *data;
318     int hdrlen, iplen;
319     int version, flow, class;
320     uchar_t proto;
321     boolean_t isfrag = B_FALSE;
322     uint8_t extmask;
323     /*
324      * The print_srcname and print_dstname strings are the hostname

```

```

326     * parts of the verbose IPv6 header output, including the comma
327     * and the space after the litteral address strings.
328     */
329     char print_srcname[MAXHOSTNAMELEN + 2];
330     char print_dstname[MAXHOSTNAMELEN + 2];
331     char src_addrstr[INET6_ADDRSTRLEN];
332     char dst_addrstr[INET6_ADDRSTRLEN];
333
334     iplen = ntohs(ip6h->ip6_plen);
335     hdrlen = IPV6_HDR_LEN;
336     fraglen -= hdrlen;
337     if (fraglen < 0)
338         return (fraglen + hdrlen);
339     data = ((uint8_t *)ip6h) + hdrlen;
340
341     proto = ip6h->ip6_nxt;
342
343     src_name = addrtoname(AF_INET6, &ip6h->ip6_src);
344     dst_name = addrtoname(AF_INET6, &ip6h->ip6_dst);
345
346     /*
347     * Use endian-aware masks to extract traffic class and
348     * flowinfo. Also, flowinfo is now 20 bits and class 8
349     * rather than 24 and 4.
350     */
351     class = ntohl((ip6h->ip6_vcf & IPV6_FLOWINFO_TCLASS) >> 20);
352     flow = ntohl(ip6h->ip6_vcf & IPV6_FLOWINFO_FLOWLABEL);
353
354     /*
355     * NOTE: the F_SUM and F_DTAIL flags are mutually exclusive,
356     * so the code within the first part of the following if statement
357     * will not affect the detailed printing of the packet.
358     */
359     if (flags & F_SUM) {
360         (void) snprintf(get_sum_line(), MAXLINE,
361             "IPv6 S=%s D=%s LEN=%d HOPS=%d CLASS=0x%x FLOW=0x%x",
362             src_name, dst_name, iplen, ip6h->ip6_hops, class, flow);
363     } else if (flags & F_DTAIL) {
364         (void) inet_ntop(AF_INET6, &ip6h->ip6_src, src_addrstr,
365             INET6_ADDRSTRLEN);
366         (void) inet_ntop(AF_INET6, &ip6h->ip6_dst, dst_addrstr,
367             INET6_ADDRSTRLEN);
368
369         version = ntohl(ip6h->ip6_vcf) >> 28;
370
371         if (strcmp(src_name, src_addrstr) == 0) {
372             print_srcname[0] = '\0';
373         } else {
374             snprintf(print_srcname, sizeof (print_srcname),
375                 "%s", src_name);
376         }
377
378         if (strcmp(dst_name, dst_addrstr) == 0) {
379             print_dstname[0] = '\0';
380         } else {
381             snprintf(print_dstname, sizeof (print_dstname),
382                 "%s", dst_name);
383         }
384
385         show_header("IPv6: ", "IPv6 Header", iplen);
386         show_space();
387
388         (void) snprintf(get_line(0, 0), get_line_remain(),
389             "Version = %d", version);
390         (void) snprintf(get_line(0, 0), get_line_remain(),

```

```

392     "Traffic Class = %d", class);
393     (void) snprintf(get_line(0, 0), get_line_remain(),
394     "Flow label = 0x%x", flow);
395     (void) snprintf(get_line(0, 0), get_line_remain(),
396     "Payload length = %d", iphlen);
397     (void) snprintf(get_line(0, 0), get_line_remain(),
398     "Next Header = %d (%s)", proto,
399     getproto(proto));
400     (void) snprintf(get_line(0, 0), get_line_remain(),
401     "Hop Limit = %d", ip6h->ip6_hops);
402     (void) snprintf(get_line(0, 0), get_line_remain(),
403     "Source address = %s%s", src_addrstr, print_srcname);
404     (void) snprintf(get_line(0, 0), get_line_remain(),
405     "Destination address = %s%s", dst_addrstr, print_dstname);
407     show_space();
408 }
410 /*
411  * Print IPv6 Extension Headers, or skip them in the summary case.
412  * Set isfrag to true if one of the extension headers encountered
413  * was a fragment header.
414  */
415 if (proto == IPPROTO_HOPOPTS || proto == IPPROTO_DSTOPTS ||
416     proto == IPPROTO_ROUTING || proto == IPPROTO_FRAGMENT) {
417     extmask = print_ipv6_extensions(flags, &data, &proto, &iplen,
418     &fraglen);
419     if ((extmask & SNOOP_FRAGMENT) != 0) {
420         isfrag = B_TRUE;
421     }
422 }
424 /*
425  * We only want to print upper layer information if this is not
426  * a fragment, or if we're printing in detail. Note that the
427  * proto variable will be set to IPPROTO_NONE if this is a fragment
428  * with a non-zero fragment offset.
429  */
430 if (!isfrag || flags & F_DTAIL) {
431     /* go to the next protocol layer */
433     switch (proto) {
434     case IPPROTO_IP:
435         break;
436     case IPPROTO_ENCAP:
437         /* LINTED: alignment */
438         (void) interpret_ip(flags, (const struct ip *)data,
439         fraglen);
440         break;
441     case IPPROTO_ICMPV6:
442         /* LINTED: alignment */
443         (void) interpret_icmpv6(flags, (icmp6_t *)data, iphlen,
444         fraglen);
445         break;
446     case IPPROTO_IGMP:
447         interpret_igmp(flags, data, iphlen, fraglen);
448         break;
449     case IPPROTO_GGP:
450         break;
451     case IPPROTO_TCP:
452         (void) interpret_tcp(flags, (struct tcphdr *)data,
453         iphlen, fraglen);
454         break;
455     case IPPROTO_ESP:
456         (void) interpret_esp(flags, data, iphlen, fraglen);
457         break;

```

```

458     case IPPROTO_AH:
459         (void) interpret_ah(flags, data, iphlen, fraglen);
460         break;
461     case IPPROTO_EGP:
462     case IPPROTO_PUP:
463         break;
464     case IPPROTO_UDP:
465         (void) interpret_udp(flags, (struct udphdr *)data,
466         iphlen, fraglen);
467         break;
468     case IPPROTO_IDP:
469     case IPPROTO_HELLO:
470     case IPPROTO_ND:
471     case IPPROTO_RAW:
472         break;
473     case IPPROTO_IPV6:
474         /* LINTED: alignment */
475         (void) interpret_ipv6(flags, (const ip6_t *)data,
476         iphlen);
477         break;
478     case IPPROTO_SCTP:
479         (void) interpret_sctp(flags, (struct sctp_hdr *)data,
480         iphlen, fraglen);
481         break;
482     case IPPROTO_OSPF:
483         interpret_ospf6(flags, data, iphlen, fraglen);
484         break;
485     case IPPROTO_DCCP:
486         (void) interpret_dccp(flags, (struct dccphdr *)data,
487         iphlen, fraglen);
488         break;
489 #endif /* ! codereview */
490     }
491 }
493     return (iplen);
494 }
496 /*
497  * ip_ext: data including the extension header.
498  * iplen: length of the data remaining in the packet.
499  * Returns a mask of IPv6 extension headers it processed.
500  */
501 uint8_t
502 print_ipv6_extensions(int flags, uint8_t **hdr, uint8_t *next, int *iplen,
503     int *fraglen)
504 {
505     uint8_t *data_ptr;
506     uchar_t proto = *next;
507     boolean_t is_extension_header;
508     struct ip6_hbh *ip6ext_hbh;
509     struct ip6_dest *ip6ext_dest;
510     struct ip6_rthdr *ip6ext_rthdr;
511     struct ip6_frag *ip6ext_frag;
512     uint32_t exthdrlen;
513     uint8_t extmask = 0;
515     if ((hdr == NULL) || (*hdr == NULL) || (next == NULL) || (iplen == 0))
516         return (0);
518     data_ptr = *hdr;
519     is_extension_header = B_TRUE;
520     while (is_extension_header) {
522         /*
523          * There must be at least enough data left to read the

```

```

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

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

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

```

```

590     }
591 }

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

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

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

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

615     while (optlen > 0) {
616         line = get_line(0, 0);
617         remain = get_line_remain();
618         len = opt[1];
619         truncstr = len > optlen ? "?" : "";
620         switch (opt[0]) {
621         case IPOPT_EOL:
622             (void) strcpy(line, " - End of option list", remain);
623             return;
624         case IPOPT_NOP:
625             (void) strcpy(line, " - No op", remain);
626             len = 1;
627             break;
628         case IPOPT_RR:
629             (void) snprintf(line, remain,
630                 " - Record route (%d bytes%s)", len, truncstr);
631             print_route(opt);
632             break;
633         case IPOPT_TS:
634             (void) snprintf(line, remain,
635                 " - Time stamp (%d bytes%s)", len, truncstr);
636             break;
637         case IPOPT_SECURITY:
638             (void) snprintf(line, remain, " - RIPS0 (%d bytes%s)",
639                 len, truncstr);
640             print_ripso(opt);
641             break;
642         case IPOPT_COMSEC:
643             (void) snprintf(line, remain, " - CIPSO (%d bytes%s)",
644                 len, truncstr);
645             print_cipso(opt);
646             break;
647         case IPOPT_LSRR:
648             (void) snprintf(line, remain,
649                 " - Loose source route (%d bytes%s)", len,
650                 truncstr);
651             print_route(opt);
652             break;
653         case IPOPT_SATID:
654             (void) snprintf(line, remain,
655                 " - SATNET Stream id (%d bytes%s)",

```

```

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

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

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

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

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

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

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

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

```

```

722     case IPPROTO_ROUTING:   return ("IPv6-Route");
723     case IPPROTO_FRAGMENT:  return ("IPv6-Frag");
724     case IPPROTO_RSVP:      return ("RSVP");
725     case IPPROTO_ENCAP:     return ("IP-in-IP");
726     case IPPROTO_AH:        return ("AH");
727     case IPPROTO_ESP:       return ("ESP");
728     case IPPROTO_ICMP:      return ("ICMP");
729     case IPPROTO_ICMPV6:    return ("ICMPv6");
730     case IPPROTO_DSTOPTS:   return ("IPv6-DstOpts");
731     case IPPROTO_IGMP:      return ("IGMP");
732     case IPPROTO_GGP:       return ("GGP");
733     case IPPROTO_TCP:       return ("TCP");
734     case IPPROTO_EGP:       return ("EGP");
735     case IPPROTO_PUP:       return ("PUP");
736     case IPPROTO_UDP:       return ("UDP");
737     case IPPROTO_IDP:       return ("IDP");
738     case IPPROTO_HELLO:     return ("HELLO");
739     case IPPROTO_ND:        return ("ND");
740     case IPPROTO_EON:       return ("EON");
741     case IPPROTO_RAW:       return ("RAW");
742     case IPPROTO_OSPF:      return ("OSPF");
743     case IPPROTO_DCCP:      return ("DCCP");
744 #endif /* !codereview */
745     default:                 return ("");
746 }
747 }

749 static void
750 prt_routing_hdr(int flags, const struct ip6_rthdr *ipv6ext_rthdr)
751 {
752     uint8_t nxt_hdr;
753     uint8_t type;
754     uint32_t len;
755     uint8_t segleft;
756     uint32_t numaddrs;
757     int i;
758     struct ip6_rthdr0 *ipv6ext_rthdr0;
759     struct in6_addr *addrs;
760     char addr[INET6_ADDRSTRLEN];

762     /* in summary mode, we don't do anything. */
763     if (flags & F_SUM) {
764         return;
765     }

767     nxt_hdr = ipv6ext_rthdr->ip6r_nxt;
768     type = ipv6ext_rthdr->ip6r_type;
769     len = 8 * (ipv6ext_rthdr->ip6r_len + 1);
770     segleft = ipv6ext_rthdr->ip6r_segleft;

772     show_header("IPv6-Route: ", "IPv6 Routing Header", 0);
773     show_space();

775     (void) snprintf(get_line(0, 0), get_line_remain(),
776         "Next header = %d (%s)", nxt_hdr, getproto(nxt_hdr));
777     (void) snprintf(get_line(0, 0), get_line_remain(),
778         "Header length = %d", len);
779     (void) snprintf(get_line(0, 0), get_line_remain(),
780         "Routing type = %d", type);
781     (void) snprintf(get_line(0, 0), get_line_remain(),
782         "Segments left = %d", segleft);

784     if (type == IPV6_RTHDR_TYPE_0) {
785         /*
786          * XXX This loop will print all addresses in the routing header,
787          * XXX not just the segments left.

```

```

788      * XXX (The header length field is twice the number of
789      * XXX addresses)
790      * XXX At some future time, we may want to change this
791      * XXX to differentiate between the hops yet to do
792      * XXX and the hops already taken.
793      */
794      /* LINTED: alignment */
795      ipv6ext_rthdr0 = (struct ip6_rthdr0 *)ipv6ext_rthdr;
796      numaddrs = ipv6ext_rthdr0->ip6r0_len / 2;
797      addrs = (struct in6_addr *) (ipv6ext_rthdr0 + 1);
798      for (i = 0; i < numaddrs; i++) {
799          (void) inet_ntop(AF_INET6, &addrs[i], addr,
800                          INET6_ADDRSTRLEN);
801          (void) snprintf(get_line(0, 0), get_line_remain(),
802                          "address[%d]=%s", i, addr);
803      }
804  }
805
806  show_space();
807  }
808
809  static void
810  prt_fragment_hdr(int flags, const struct ip6_frag *ipv6ext_frag)
811  {
812      boolean_t morefrag;
813      uint16_t fragoffset;
814      uint8_t nxt_hdr;
815      uint32_t fragident;
816
817      /* extract the various fields from the fragment header */
818      nxt_hdr = ipv6ext_frag->ip6f_nxt;
819      morefrag = (ipv6ext_frag->ip6f_offlg & IP6F_MORE_FRAG) == 0
820          ? B_FALSE : B_TRUE;
821      fragoffset = ntohs(ipv6ext_frag->ip6f_offlg & IP6F_OFF_MASK);
822      fragident = ntohl(ipv6ext_frag->ip6f_ident);
823
824      if (flags & F_SUM) {
825          (void) snprintf(get_sum_line(), MAXLINE,
826                          "IPv6 fragment ID=%u Offset=%-4d MF=%d",
827                          fragident,
828                          fragoffset,
829                          morefrag);
830      } else { /* F_DTAIL */
831          show_header("IPv6-Frag: ", "IPv6 Fragment Header", 0);
832          show_space();
833
834          (void) snprintf(get_line(0, 0), get_line_remain(),
835                          "Next Header = %d (%s)", nxt_hdr, getproto(nxt_hdr));
836          (void) snprintf(get_line(0, 0), get_line_remain(),
837                          "Fragment Offset = %d", fragoffset);
838          (void) snprintf(get_line(0, 0), get_line_remain(),
839                          "More Fragments Flag = %s", morefrag ? "true" : "false");
840          (void) snprintf(get_line(0, 0), get_line_remain(),
841                          "Identification = %u", fragident);
842
843          show_space();
844      }
845  }
846
847  static void
848  print_ip6opt_ls(const uchar_t *data, unsigned int op_len)
849  {
850      uint32_t doi;
851      uint8_t sotype, solen;
852      uint16_t value, value2;
853      char *cp;

```

```

854      int remlen;
855      boolean_t printed;
856
857      (void) snprintf(get_line(0, 0), get_line_remain(),
858                      "Labeled Security Option len = %u bytes%s", op_len,
859                      op_len < sizeof (uint32_t) || (op_len & 1) != 0 ? "?" : "");
860      if (op_len < sizeof (uint32_t))
861          return;
862      GETINT32(doi, data);
863      (void) snprintf(get_line(0, 0), get_line_remain(),
864                      "    DOI = %d (%s)", doi, doi == IP6LS_DOI_V4 ? "IPv4" : "???");
865      op_len -= sizeof (uint32_t);
866      while (op_len > 0) {
867          GETINT8(sotype, data);
868          if (op_len < 2) {
869              (void) snprintf(get_line(0, 0), get_line_remain(),
870                              "        truncated %u suboption (no len)", sotype);
871              break;
872          }
873          GETINT8(solen, data);
874          if (solen < 2 || solen > op_len) {
875              (void) snprintf(get_line(0, 0), get_line_remain(),
876                              "        bad %u suboption (len 2 <= %u <= %u)",
877                              sotype, solen, op_len);
878              if (solen < 2)
879                  solen = 2;
880              if (solen > op_len)
881                  solen = op_len;
882          }
883          op_len -= solen;
884          solen -= 2;
885          cp = get_line(0, 0);
886          remlen = get_line_remain();
887          (void) strcpy(cp, "    ", remlen);
888          cp += 4;
889          remlen -= 4;
890          printed = B_TRUE;
891          switch (sotype) {
892              case IP6LS_TT_LEVEL:
893                  if (solen != 2) {
894                      printed = B_FALSE;
895                      break;
896                  }
897                  GETINT16(value, data);
898                  (void) snprintf(cp, remlen, "Level %u", value);
899                  solen = 0;
900                  break;
901              case IP6LS_TT_VECTOR:
902                  (void) strcpy(cp, "Bit-Vector: ", remlen);
903                  remlen -= strlen(cp);
904                  cp += strlen(cp);
905                  while (solen > 1) {
906                      GETINT16(value, data);
907                      solen -= 2;
908                      (void) snprintf(cp, remlen, "%04x", value);
909                      remlen -= strlen(cp);
910                      cp += strlen(cp);
911                  }
912                  break;
913              case IP6LS_TT_ENUM:
914                  (void) strcpy(cp, "Enumeration:", remlen);
915                  remlen -= strlen(cp);
916                  cp += strlen(cp);
917                  while (solen > 1) {
918                      GETINT16(value, data);
919                      solen -= 2;

```

```

920         (void) snprintf(cp, remlen, " %u", value);
921         remlen -= strlen(cp);
922         cp += strlen(cp);
923     }
924     break;
925 case IP6LS_TT_RANGES:
926     (void) strcpy(cp, "Ranges:", remlen);
927     remlen -= strlen(cp);
928     cp += strlen(cp);
929     while (solen > 3) {
930         GETINT16(value, data);
931         GETINT16(value2, data);
932         solen -= 4;
933         (void) snprintf(cp, remlen, " %u-%u", value,
934             value2);
935         remlen -= strlen(cp);
936         cp += strlen(cp);
937     }
938     break;
939 case IP6LS_TT_V4:
940     (void) strcpy(cp, "IPv4 Option", remlen);
941     print_ipoptions(data, solen);
942     solen = 0;
943     break;
944 case IP6LS_TT_DEST:
945     (void) snprintf(cp, remlen,
946         "Destination-Only Data length %u", solen);
947     solen = 0;
948     break;
949 default:
950     (void) snprintf(cp, remlen,
951         " unknown %u suboption (len %u)", sotype, solen);
952     solen = 0;
953     break;
954 }
955 if (solen != 0) {
956     if (printed) {
957         cp = get_line(0, 0);
958         remlen = get_line_remain();
959     }
960     (void) snprintf(cp, remlen,
961         " malformed %u suboption (remaining %u)",
962         sotype, solen);
963     data += solen;
964 }
965 }
966 }

968 static void
969 prt_hbh_options(int flags, const struct ip6_hbh *ipv6ext_hbh)
970 {
971     const uint8_t *data, *ndata;
972     uint32_t len;
973     uint8_t op_type;
974     uint8_t op_len;
975     uint8_t nxt_hdr;

977     /* in summary mode, we don't do anything. */
978     if (flags & F_SUM) {
979         return;
980     }

982     show_header("IPv6-HopOpts: ", "IPv6 Hop-by-Hop Options Header", 0);
983     show_space();

985     /*

```

```

986     * Store the length of this ext_hdr in bytes. The caller has
987     * ensured that there is at least len bytes of data left.
988     */
989     len = ipv6ext_hbh->ip6h_len * 8 + 8;

991     ndata = (const uint8_t *)ipv6ext_hbh + 2;
992     len -= 2;

994     nxt_hdr = ipv6ext_hbh->ip6h_nxt;
995     (void) snprintf(get_line(0, 0), get_line_remain(),
996         "Next Header = %u (%s)", nxt_hdr, getproto(nxt_hdr));

998     while (len > 0) {
999         data = ndata;
1000         GETINT8(op_type, data);
1001         /* This is the only one-octet IPv6 option */
1002         if (op_type == IP6OPT_PAD1) {
1003             (void) snprintf(get_line(0, 0), get_line_remain(),
1004                 "pad1 option ");
1005             len--;
1006             ndata = data;
1007             continue;
1008         }
1009         GETINT8(op_len, data);
1010         if (len < 2 || op_len + 2 > len) {
1011             (void) snprintf(get_line(0, 0), get_line_remain(),
1012                 "Error: option %u truncated (%u + 2 > %u)",
1013                 op_type, op_len, len);
1014             op_len = len - 2;
1015             /*
1016              * Continue processing the malformed option so that we
1017              * can display as much as possible.
1018              */
1019         }

1021         /* advance pointers to the next option */
1022         len -= op_len + 2;
1023         ndata = data + op_len;

1025         /* process this option */
1026         switch (op_type) {
1027         case IP6OPT_PADN:
1028             (void) snprintf(get_line(0, 0), get_line_remain(),
1029                 "padN option len = %u", op_len);
1030             break;
1031         case IP6OPT_JUMBO: {
1032             uint32_t payload_len;

1034             (void) snprintf(get_line(0, 0), get_line_remain(),
1035                 "Jumbo Payload Option len = %u bytes%s", op_len,
1036                 op_len == sizeof(uint32_t) ? "" : "?");
1037             if (op_len == sizeof(uint32_t)) {
1038                 GETINT32(payload_len, data);
1039                 (void) snprintf(get_line(0, 0),
1040                     get_line_remain(),
1041                     "Jumbo Payload Length = %u bytes",
1042                     payload_len);
1043             }
1044             break;
1045         }
1046         case IP6OPT_ROUTER_ALERT: {
1047             uint16_t value;
1048             const char *label[] = {"MLD", "RSVP", "AN"};

1050             (void) snprintf(get_line(0, 0), get_line_remain(),
1051                 "Router Alert Option len = %u bytes%s", op_len,

```

```

1052         op_len == sizeof (uint16_t) ? "" : "?");
1053         if (op_len == sizeof (uint16_t)) {
1054             GETINT16(value, data);
1055             (void) snprintf(get_line(0, 0),
1056                 get_line_remain(),
1057                 "Alert Type = %d (%s)", value,
1058                 value < sizeof (label) / sizeof (label[0]) ?
1059                 label[value] : "???");
1060         }
1061         break;
1062     }
1063     case IP6OPT_LS:
1064         print_ip6opt_ls(data, op_len);
1065         break;
1066     default:
1067         (void) snprintf(get_line(0, 0), get_line_remain(),
1068             "Option type = %u, len = %u", op_type, op_len);
1069         break;
1070 }
1071 }

1073     show_space();
1074 }

1076 static void
1077 prt_dest_options(int flags, const struct ip6_dest *ipv6ext_dest)
1078 {
1079     const uint8_t *data, *ndata;
1080     uint32_t len;
1081     uint8_t op_type;
1082     uint32_t op_len;
1083     uint8_t nxt_hdr;
1084     uint8_t value;

1086     /* in summary mode, we don't do anything. */
1087     if (flags & F_SUM) {
1088         return;
1089     }

1091     show_header("IPv6-DstOpts:  ", "IPv6 Destination Options Header", 0);
1092     show_space();

1094     /*
1095      * Store the length of this ext_hdr in bytes. The caller has
1096      * ensured that there is at least len bytes of data left.
1097      */
1098     len = ipv6ext_dest->ip6d_len * 8 + 8;

1100     ndata = (const uint8_t *)ipv6ext_dest + 2; /* skip hdr/len */
1101     len -= 2;

1103     nxt_hdr = ipv6ext_dest->ip6d_nxt;
1104     (void) snprintf(get_line(0, 0), get_line_remain(),
1105         "Next Header = %u (%s)", nxt_hdr, getproto(nxt_hdr));

1107     while (len > 0) {
1108         data = ndata;
1109         GETINT8(op_type, data);
1110         if (op_type == IP6OPT_PAD1) {
1111             (void) snprintf(get_line(0, 0), get_line_remain(),
1112                 "pad1 option ");
1113             len--;
1114             ndata = data;
1115             continue;
1116         }
1117         GETINT8(op_len, data);

```

```

1118         if (len < 2 || op_len + 2 > len) {
1119             (void) snprintf(get_line(0, 0), get_line_remain(),
1120                 "Error: option %u truncated (%u + 2 > %u)",
1121                 op_type, op_len, len);
1122             op_len = len - 2;
1123             /*
1124              * Continue processing the malformed option so that we
1125              * can display as much as possible.
1126              */
1127         }

1129         /* advance pointers to the next option */
1130         len -= op_len + 2;
1131         ndata = data + op_len;

1133         /* process this option */
1134         switch (op_type) {
1135             case IP6OPT_PADN:
1136                 (void) snprintf(get_line(0, 0), get_line_remain(),
1137                     "padN option len = %u", op_len);
1138                 break;
1139             case IP6OPT_TUNNEL_LIMIT:
1140                 GETINT8(value, data);
1141                 (void) snprintf(get_line(0, 0), get_line_remain(),
1142                     "tunnel encapsulation limit len = %d, value = %d",
1143                     op_len, value);
1144                 break;
1145             case IP6OPT_LS:
1146                 print_ip6opt_ls(data, op_len);
1147                 break;
1148             default:
1149                 (void) snprintf(get_line(0, 0), get_line_remain(),
1150                     "Option type = %u, len = %u", op_type, op_len);
1151                 break;
1152         }
1153     }

1155     show_space();
1156 }

1158 #define ALABEL_MAXLEN    256

1160 static char ascii_label[ALABEL_MAXLEN];
1161 static char *plabel = ascii_label;

1163 struct snoop_pair {
1164     int val;
1165     const char *name;
1166 };

1168 static struct snoop_pair ripso_class_tbl[] = {
1169     TSOL_CL_TOP_SECRET,    "TOP SECRET",
1170     TSOL_CL_SECRET,        "SECRET",
1171     TSOL_CL_CONFIDENTIAL,   "CONFIDENTIAL",
1172     TSOL_CL_UNCLASSIFIED,   "UNCLASSIFIED",
1173     -1,                     NULL
1174 };

1176 static struct snoop_pair ripso_prot_tbl[] = {
1177     TSOL_PA_GENSER,        "GENSER",
1178     TSOL_PA_SIOP_ESI,      "SIOP-ESI",
1179     TSOL_PA_SCI,           "SCI",
1180     TSOL_PA_NSA,           "NSA",
1181     TSOL_PA_DOE,           "DOE",
1182     0x04,                  "UNASSIGNED",
1183     0x02,                  "UNASSIGNED",

```



```

1184     -1,                NULL
1185 };

1187 static struct snoop_pair *
1188 get_pair_byval(struct snoop_pair pairlist[], int val)
1189 {
1190     int i;

1192     for (i = 0; pairlist[i].name != NULL; i++)
1193         if (pairlist[i].val == val)
1194             return (&pairlist[i]);
1195     return (NULL);
1196 }

1198 static void
1199 print_ripso(const uchar_t *opt)
1200 {
1201     struct snoop_pair *ripso_class;
1202     int i, index, prot_len;
1203     boolean_t first_prot;
1204     char line[100], *ptr;

1206     prot_len = opt[1] - 3;
1207     if (prot_len < 0)
1208         return;

1210     show_header("RIPSO: ", "Revised IP Security Option", 0);
1211     show_space();

1213     (void) snprintf(get_line(0, 0), get_line_remain(),
1214         "Type = Basic Security Option (%d), Length = %d", opt[0], opt[1]);

1216     /*
1217      * Display Classification Level
1218      */
1219     ripso_class = get_pair_byval(ripso_class_tbl, (int)opt[2]);
1220     if (ripso_class != NULL)
1221         (void) snprintf(get_line(0, 0), get_line_remain(),
1222             "Classification = Unknown (0x%02x)", opt[2]);
1223     else
1224         (void) snprintf(get_line(0, 0), get_line_remain(),
1225             "Classification = %s (0x%02x)",
1226             ripso_class->name, ripso_class->val);

1228     /*
1229      * Display Protection Authority Flags
1230      */
1231     (void) snprintf(line, sizeof (line), "Protection Authority = ");
1232     ptr = line;
1233     first_prot = B_TRUE;
1234     for (i = 0; i < prot_len; i++) {
1235         index = 0;
1236         while (ripso_prot_tbl[index].name != NULL) {
1237             if (opt[3 + i] & ripso_prot_tbl[index].val) {
1238                 ptr = strchr(ptr, 0);
1239                 if (!first_prot) {
1240                     (void) strlcpy(ptr, " ",
1241                         sizeof (line) - (ptr - line));
1242                     ptr = strchr(ptr, 0);
1243                 }
1244                 (void) snprintf(ptr,
1245                     sizeof (line) - (ptr - line),
1246                     "%s (0x%02x)",
1247                     ripso_prot_tbl[index].name,
1248                     ripso_prot_tbl[index].val);
1249             }

```

```

1250         index++;
1251     }
1252     if ((opt[3 + i] & 1) == 0)
1253         break;
1254 }
1255 if (!first_prot)
1256     (void) snprintf(get_line(0, 0), get_line_remain(), "%s", line);
1257 else
1258     (void) snprintf(get_line(0, 0), get_line_remain(), "%sNone",
1259         line);
1260 }

1262 #define CIPSO_GENERIC_ARRAY_LEN 200

1264 /*
1265  * Return 1 if CIPSO SL and Categories are all 1's; 0 otherwise.
1266  * Note: opt starts with "Tag Type":
1267  * |tag_type(1)|tag_length(1)|align(1)|sl(1)|categories(variable)|
1268  */
1269 static boolean_t
1270 cipso_high(const uchar_t *opt)
1271 {
1272     int i;

1274     if (((int)opt[1] + 6) < IP_MAX_OPT_LENGTH)
1275         return (B_FALSE);
1276     for (i = 0; i < ((int)opt[1] - 3); i++)
1277         if (opt[3 + i] != 0xff)
1278             return (B_FALSE);
1279     return (B_TRUE);
1280 }

1282 /*
1283  * Converts CIPSO label to SL.
1284  * Note: opt starts with "Tag Type":
1285  * |tag_type(1)|tag_length(1)|align(1)|sl(1)|categories(variable)|
1286  */
1287 static void
1288 cipso2sl(const uchar_t *opt, bslabel_t *sl, int *high)
1289 {
1290     int i, taglen;
1291     uchar_t *q = (uchar_t *)&((bslabel_impl_t *)sl)->compartments;

1293     *high = 0;
1294     taglen = opt[1];
1295     memset((caddr_t)sl, 0, sizeof (bslabel_t));

1297     if (cipso_high(opt)) {
1298         BSLHIGH(sl);
1299         *high = 1;
1300     } else {
1301         LCLASS_SET((bslabel_impl_t *)sl, opt[3]);
1302         for (i = 0; i < taglen - TSOL_TTL_MIN_LENGTH; i++)
1303             q[i] = opt[TSOL_TTL_MIN_LENGTH + i];
1304     }
1305     SETBLTYPE(sl, SUN_SL_ID);
1306 }

1308 static int
1309 interpret_cipso_tagtype1(const uchar_t *opt)

```

```

1316 {
1317     int i, taglen, ishigh;
1318     bslabel_t sl;
1319     char line[CIPSO_GENERIC_ARRAY_LEN], *ptr;

1321     taglen = opt[1];
1322     if (taglen < TSOL_TT1_MIN_LENGTH ||
1323         taglen > TSOL_TT1_MAX_LENGTH)
1324         return (taglen);

1326     (void) snprintf(get_line(0, 0), get_line_remain(),
1327         "Tag Type = %d, Tag Length = %d", opt[0], opt[1]);
1328     (void) snprintf(get_line(0, 0), get_line_remain(),
1329         "Sensitivity Level = 0x%02x", opt[3]);
1330     ptr = line;
1331     for (i = 0; i < taglen - TSOL_TT1_MIN_LENGTH; i++) {
1332         (void) snprintf(ptr, sizeof (line) - (ptr - line), "%02x",
1333             opt[TSOL_TT1_MIN_LENGTH + i]);
1334         ptr = strchr(ptr, 0);
1335     }
1336     if (i != 0) {
1337         (void) snprintf(get_line(0, 0), get_line_remain(),
1338             "Categories = ");
1339         (void) snprintf(get_line(0, 0), get_line_remain(), "\t%s",
1340             line);
1341     } else {
1342         (void) snprintf(get_line(0, 0), get_line_remain(),
1343             "Categories = None");
1344     }
1345     cipso2sl(opt, &sl, &ishigh);
1346     if (is_system_labeled()) {
1347         if (bsltos(&sl, &labeled, ALABEL_MAXLEN,
1348             LONG_CLASSIFICATION|LONG_WORDS|VIEW_INTERNAL) < 0) {
1349             (void) snprintf(get_line(0, 0), get_line_remain(),
1350                 "The Sensitivity Level and Categories can't be "
1351                 "mapped to a valid SL");
1352         } else {
1353             (void) snprintf(get_line(0, 0), get_line_remain(),
1354                 "The Sensitivity Level and Categories are mapped "
1355                 "to the SL:");
1356             (void) snprintf(get_line(0, 0), get_line_remain(),
1357                 "\t%s", ascii_label);
1358         }
1359     }
1360     return (taglen);
1361 }

1363 /*
1364  * The following struct definition #define's are copied from TS1.x. They are
1365  * not used here (except TTYPE_3_MAX_TOKENS), but included as a reference for
1366  * the tag type 3 packet format.
1367  */
1368 #define TTYPE_3_MAX_TOKENS      7

1370 /*
1371  * Display CIPSO tag type 3 which is defined by MAXSIX.
1372  */
1373 static int
1374 interpret_cipso_tagtype3(const uchar_t *opt)
1375 {
1376     uchar_t tagtype;
1377     int index, numtokens, taglen;
1378     uint16_t mask;
1379     uint32_t token;
1380     static const char *name[] = {
1381         "SL",

```

```

1382         "NCAV",
1383         "INTEG",
1384         "SID",
1385         "undefined",
1386         "undefined",
1387         "IL",
1388         "PRIVS",
1389         "LUID",
1390         "PID",
1391         "IDS",
1392         "ACL"
1393     };

1395     tagtype = *opt++;
1396     (void) memcpy(&mask, opt + 3, sizeof (mask));
1397     (void) snprintf(get_line(0, 0), get_line_remain(),
1398         "Tag Type = %d (MAXSIX)", tagtype);
1399     (void) snprintf(get_line(0, 0), get_line_remain(),
1400         "Generation = 0x%02x%02x%02x, Mask = 0x%04x", opt[0], opt[1],
1401         opt[2], mask);
1402     opt += 3 + sizeof (mask);

1404     /*
1405      * Display tokens
1406      */
1407     numtokens = 0;
1408     index = 0;
1409     while (mask != 0 && numtokens < TTYPE_3_MAX_TOKENS) {
1410         if (mask & 0x0001) {
1411             (void) memcpy(&token, opt, sizeof (token));
1412             opt += sizeof (token);
1413             (void) snprintf(get_line(0, 0), get_line_remain(),
1414                 "Attribute = %s, Token = 0x%08x",
1415                 index < sizeof (name) / sizeof (*name) ?
1416                 name[index] : "unknown", token);
1417             numtokens++;
1418         }
1419         mask = mask >> 1;
1420         index++;
1421     }

1423     taglen = 6 + numtokens * 4;
1424     return (taglen);
1425 }

1427 static void
1428 print_cipso(const uchar_t *opt)
1429 {
1430     int optlen, taglen, tagnum;
1431     uint32_t doi;
1432     char line[CIPSO_GENERIC_ARRAY_LEN];
1433     char *oldnest;

1435     optlen = opt[1];
1436     if (optlen < TSOL_CIPSO_MIN_LENGTH || optlen > TSOL_CIPSO_MAX_LENGTH)
1437         return;

1439     oldnest = prot_nest_prefix;
1440     prot_nest_prefix = prot_prefix;
1441     show_header("CIPSO: ", "Common IP Security Option", 0);
1442     show_space();

1444     /*
1445      * Display CIPSO Header
1446      */
1447     (void) snprintf(get_line(0, 0), get_line_remain(),

```

```
1448     "Type = CIPSO (%d), Length = %d", opt[0], opt[1]);
1449     (void) memcpy(&doi, opt + 2, sizeof (doi));
1450     (void) snprintf(get_line(0, 0), get_line_remain(),
1451     "Domain of Interpretation = %u", (unsigned)ntohl(doi));
1452
1453     if (opt[1] == TSOL_CIPSO_MIN_LENGTH) { /* no tags */
1454         show_space();
1455         prot_prefix = prot_nest_prefix;
1456         prot_nest_prefix = oldnest;
1457         return;
1458     }
1459     optlen -= TSOL_CIPSO_MIN_LENGTH;
1460     opt += TSOL_CIPSO_MIN_LENGTH;
1461
1462     /*
1463     * Display Each Tag
1464     */
1465     tagnum = 1;
1466     while (optlen >= TSOL_TT1_MIN_LENGTH) {
1467         (void) snprintf(line, sizeof (line), "Tag# %d", tagnum);
1468         show_header("CIPSO: ", line, 0);
1469         /*
1470         * We handle tag type 1 and 3 only. Note, tag type 3
1471         * is MAXSIX defined.
1472         */
1473         switch (opt[0]) {
1474             case 1:
1475                 taglen = interpret_cipso_tagtype1(opt);
1476                 break;
1477             case 3:
1478                 taglen = interpret_cipso_tagtype3(opt);
1479                 break;
1480             default:
1481                 (void) snprintf(get_line(0, 0), get_line_remain(),
1482                 "Unknown Tag Type %d", opt[0]);
1483                 show_space();
1484                 prot_prefix = prot_nest_prefix;
1485                 prot_nest_prefix = oldnest;
1486                 return;
1487         }
1488
1489         /*
1490         * Move to the next tag
1491         */
1492         if (taglen <= 0)
1493             break;
1494         optlen -= taglen;
1495         opt += taglen;
1496         tagnum++;
1497     }
1498     show_space();
1499     prot_prefix = prot_nest_prefix;
1500     prot_nest_prefix = oldnest;
1501 }
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ipsec.c

1

6896 Sat Aug 18 10:36:58 2012

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ipsec.c

dccp: snoop, build system fixes

_____unchanged_portion_omitted_____

```
102 int
103 interpret_ah(int flags, uint8_t *hdr, int iplen, int fraglen)
104 {
105     /* LINTED: alignment */
106     ah_t *ah = (ah_t *)hdr;
107     ah_t *aligned_ah;
108     ah_t storage; /* In case hdr isn't aligned. */
109     char *line, *buff;
110     uint_t ahlen, auth_data_len;
111     uint8_t *auth_data, *data;
112     int new_iphlen;
113     uint8_t proto;
114
115     if (fraglen < sizeof (ah_t))
116         return (fraglen); /* incomplete header */
117
118     if (!IS_P2ALIGNED(hdr, 4)) {
119         aligned_ah = (ah_t *)&storage;
120         bcopy(hdr, &storage, sizeof (ah_t));
121     } else {
122         aligned_ah = ah;
123     }
124
125     /*
126      * "+ 8" is for the "constant" part that's not included in the AH
127      * length.
128      *
129      * The AH RFC specifies the length field in "length in 4-byte units,
130      * not counting the first 8 bytes". So if an AH is 24 bytes long,
131      * the length field will contain "4". (4 * 4 + 8 == 24).
132      */
133     ahlen = (aligned_ah->ah_length << 2) + 8;
134     fraglen -= ahlen;
135     if (fraglen < 0)
136         return (fraglen + ahlen); /* incomplete header */
137
138     auth_data_len = ahlen - sizeof (ah_t);
139     auth_data = (uint8_t *) (ah + 1);
140     data = auth_data + auth_data_len;
141
142     if (flags & F_SUM) {
143         line = (char *)get_sum_line();
144         (void) sprintf(line, "AH SPI=0x%x Replay=%u",
145             ntohl(aligned_ah->ah_spi), ntohl(aligned_ah->ah_replay));
146         line += strlen(line);
147     }
148
149     if (flags & F_DTAIL) {
150         show_header("AH: ", "Authentication Header", ahlen);
151         show_space();
152         (void) sprintf(get_line((char *)&ah->ah_nexthdr - dlc_header,
153             1), "Next header = %d (%s)", aligned_ah->ah_nexthdr,
154             getproto(aligned_ah->ah_nexthdr));
155         (void) sprintf(get_line((char *)&ah->ah_length - dlc_header, 1),
156             "AH length = %d (%d bytes)", aligned_ah->ah_length, ahlen);
157         (void) sprintf(get_line((char *)&ah->ah_reserved - dlc_header,
158             2), "<Reserved field = 0x%x>",
159             ntohs(aligned_ah->ah_reserved));
160         (void) sprintf(get_line((char *)&ah->ah_spi - dlc_header, 4),
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_ipsec.c

2

```
161     "SPI = 0x%x", ntohl(aligned_ah->ah_spi));
162     (void) sprintf(get_line((char *)&ah->ah_replay - dlc_header, 4),
163         "Replay = %u", ntohl(aligned_ah->ah_replay));
164
165     /*
166      * 2 for two hex digits per auth_data byte
167      * plus one byte for trailing null byte.
168      */
169     buff = malloc(auth_data_len * 2 + 1);
170     if (buff != NULL) {
171         int i;
172
173         for (i = 0; i < auth_data_len; i++)
174             sprintf(buff + i * 2, "%02x", auth_data[i]);
175     }
176
177     (void) sprintf(get_line((char *)auth_data - dlc_header,
178         auth_data_len), "ICV = %s",
179         (buff == NULL) ? "<out of memory>" : buff);
180
181     /* malloc(3c) says I can call free even if buff == NULL */
182     free(buff);
183
184     show_space();
185 }
186
187 new_iphlen = iplen - ahlen;
188 proto = aligned_ah->ah_nexthdr;
189
190 /*
191  * Print IPv6 Extension Headers, or skip them in the summary case.
192  */
193 if (proto == IPPROTO_HOPOPTS || proto == IPPROTO_DSTOPTS ||
194     proto == IPPROTO_ROUTING || proto == IPPROTO_FRAGMENT) {
195     (void) print_ipv6_extensions(flags, &data, &proto, &iplen,
196         &fraglen);
197 }
198
199 if (fraglen > 0)
200     switch (proto) {
201     case IPPROTO_ENCAP:
202         /* LINTED: alignment */
203         (void) interpret_ip(flags, (struct ip *)data,
204             new_iphlen);
205         break;
206     case IPPROTO_IPV6:
207         (void) interpret_ipv6(flags, (ip6_t *)data,
208             new_iphlen);
209         break;
210     case IPPROTO_ICMP:
211         (void) interpret_icmp(flags,
212             /* LINTED: alignment */
213             (struct icmp *)data, new_iphlen, fraglen);
214         break;
215     case IPPROTO_ICMPV6:
216         /* LINTED: alignment */
217         (void) interpret_icmpv6(flags, (icmp6_t *)data,
218             new_iphlen, fraglen);
219         break;
220     case IPPROTO_TCP:
221         (void) interpret_tcp(flags,
222             (struct tcphdr *)data, new_iphlen, fraglen);
223         break;
224
225     case IPPROTO_ESP:
226         (void) interpret_esp(flags, data, new_iphlen,
```

```
227             fraglen);
228             break;
229
230         case IPPROTO_AH:
231             (void) interpret_ah(flags, data, new_iphlen,
232             fraglen);
233             break;
234
235         case IPPROTO_UDP:
236             (void) interpret_udp(flags,
237             (struct udphdr *)data, new_iphlen, fraglen);
238             break;
239         case IPPROTO_DCCP:
240             (void) interpret_dccp(flags,
241             (struct dccphdr *)data, new_iphlen, fraglen);
242             break;
243 #endif /* ! codereview */
244         /* default case is to not print anything else */
245     }
246
247     return (ahlen);
248 }
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_pf.c

1

39796 Sat Aug 18 10:36:59 2012

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_pf.c

dccp: options and features

_____unchanged portion omitted_____

```
140 static transport_table_t ether_transport_mapping_table[] = {
141     {IPPROTO_TCP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
142     {IPPROTO_TCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
143     {IPPROTO_UDP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
144     {IPPROTO_UDP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
145     {IPPROTO OSPF, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
146     {IPPROTO OSPF, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
147     {IPPROTO_SCTP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
148     {IPPROTO_SCTP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
149     {IPPROTO_ICMP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
150     {IPPROTO_ICMPV6, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
151     {IPPROTO_ENCAP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
152     {IPPROTO_ESP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
153     {IPPROTO_ESP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
154     {IPPROTO_AH, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
155     {IPPROTO_AH, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
156     {IPPROTO_DCCP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
157     {IPPROTO_DCCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
158 #endif /* ! codereview */
159     {-1, 0, 0} /* must be the final entry */
160 };

162 static transport_table_t ipnet_transport_mapping_table[] = {
163     {IPPROTO_TCP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
164       IPV4_TYPE_HEADER_OFFSET},
165     {IPPROTO_TCP, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
166       IPV6_TYPE_HEADER_OFFSET},
167     {IPPROTO_UDP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
168       IPV4_TYPE_HEADER_OFFSET},
169     {IPPROTO_UDP, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
170       IPV6_TYPE_HEADER_OFFSET},
171     {IPPROTO OSPF, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
172       IPV4_TYPE_HEADER_OFFSET},
173     {IPPROTO OSPF, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
174       IPV6_TYPE_HEADER_OFFSET},
175     {IPPROTO_SCTP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
176       IPV4_TYPE_HEADER_OFFSET},
177     {IPPROTO_SCTP, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
178       IPV6_TYPE_HEADER_OFFSET},
179     {IPPROTO_ICMP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
180       IPV4_TYPE_HEADER_OFFSET},
181     {IPPROTO_ICMPV6, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
182       IPV6_TYPE_HEADER_OFFSET},
183     {IPPROTO_ENCAP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
184       IPV4_TYPE_HEADER_OFFSET},
185     {IPPROTO_ESP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
186       IPV4_TYPE_HEADER_OFFSET},
187     {IPPROTO_ESP, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
188       IPV6_TYPE_HEADER_OFFSET},
189     {IPPROTO_AH, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
190       IPV4_TYPE_HEADER_OFFSET},
191     {IPPROTO_AH, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
192       IPV6_TYPE_HEADER_OFFSET},
193     {IPPROTO_DCCP, (DL_IPNETINFO_VERSION <= 8 | AF_INET),
194       IPV4_TYPE_HEADER_OFFSET},
195     {IPPROTO_DCCP, (DL_IPNETINFO_VERSION <= 8 | AF_INET6),
196       IPV6_TYPE_HEADER_OFFSET},
197 #endif /* ! codereview */
198     {-1, 0, 0} /* must be the final entry */
```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_pf.c

2

199 };

```
201 static transport_table_t ib_transport_mapping_table[] = {
202     {IPPROTO_TCP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
203     {IPPROTO_TCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
204     {IPPROTO_UDP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
205     {IPPROTO_UDP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
206     {IPPROTO OSPF, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
207     {IPPROTO OSPF, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
208     {IPPROTO_SCTP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
209     {IPPROTO_SCTP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
210     {IPPROTO_ICMP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
211     {IPPROTO_ICMPV6, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
212     {IPPROTO_ENCAP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
213     {IPPROTO_ESP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
214     {IPPROTO_ESP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
215     {IPPROTO_AH, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
216     {IPPROTO_AH, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
217     {IPPROTO_DCCP, ETHERTYPE_IP, IPV4_TYPE_HEADER_OFFSET},
218     {IPPROTO_DCCP, ETHERTYPE_IPV6, IPV6_TYPE_HEADER_OFFSET},
219 #endif /* ! codereview */
220     {-1, 0, 0} /* must be the final entry */
221 };

223 typedef struct datalink {
224     uint_t          dl_type;
225     void            (*dl_match_fn)(uint_t datatype);
226     transport_table_t *dl_trans_map_tbl;
227     network_table_t *dl_net_map_tbl;
228     int             dl_link_header_len;
229     int             dl_link_type_offset;
230     int             dl_link_dest_offset;
231     int             dl_link_src_offset;
232     int             dl_link_addr_len;
233 } datalink_t;

235 datalink_t          dl;

237 #define IPV4_SRCADDR_OFFSET (dl.dl_link_header_len + 12)
238 #define IPV4_DSTADDR_OFFSET (dl.dl_link_header_len + 16)
239 #define IPV6_SRCADDR_OFFSET (dl.dl_link_header_len + 8)
240 #define IPV6_DSTADDR_OFFSET (dl.dl_link_header_len + 24)

242 #define IPNET_SRCZONE_OFFSET 16
243 #define IPNET_DSTZONE_OFFSET 20

245 static int inBrace = 0, inBraceOR = 0;
246 static int foundOR = 0;
247 char *tkp, *sav_tkp;
248 char *token;
249 enum { EOL, ALPHA, NUMBER, FIELD, ADDR_IP, ADDR_ETHER, SPECIAL,
250        ADDR_IP6 } tokentype;
251 uint_t tokenval;

253 enum direction { ANY, TO, FROM };
254 enum direction dir;

256 extern void next();

258 static void pf_expression();
259 static void pf_check_vlan_tag(uint_t offset);
260 static void pf_clear_offset_register();
261 static void pf_emit_load_offset(uint_t offset);
262 static void pf_match_ethertype(uint_t ethertype);
263 static void pf_match_ipnettype(uint_t type);
264 static void pf_match_ibtype(uint_t type);
```

```

265 static void pf_check_transport_protocol(uint_t transport_protocol);
266 static void pf_compare_value_mask_generic(int offset, uint_t len,
267     uint_t val, int mask, uint_t op);
268 static void pf_matchfn(const char *name);

270 /*
271  * This pointer points to the function that last generated
272  * instructions to change the offset register. It's used
273  * for comparisons to see if we need to issue more instructions
274  * to change the register.
275  *
276  * It's initialized to pf_clear_offset_register because the offset
277  * register in pfmod is initialized to zero, similar to the state
278  * it would be in after executing the instructions issued by
279  * pf_clear_offset_register.
280  */
281 static void *last_offset_operation = (void*)pf_clear_offset_register;

283 static void
284 pf_emit(x)
285     ushort_t x;
286 {
287     if (pfp > &pf.Pf_Filter[PF_MAXFILTERS - 1])
288         longjmp(env, 1);
289     *pfp++ = x;
290 }

292 static void
293 pf_codeprint(code, len)
294     ushort_t *code;
295     int len;
296 {
297     ushort_t *pc;
298     ushort_t *plast = code + len;
299     int op, action;

301     if (len > 0) {
302         printf("Kernel Filter:\n");
303     }

305     for (pc = code; pc < plast; pc++) {
306         printf("\t%3d: ", pc - code);

308         op = *pc & 0xfc00; /* high 10 bits */
309         action = *pc & 0x3ff; /* low 6 bits */

311         switch (action) {
312             case ENF_PUSHLIT:
313                 printf("PUSHLIT ");
314                 break;
315             case ENF_PUSHZERO:
316                 printf("PUSHZERO ");
317                 break;
318             #ifdef ENF_PUSHONE
319                 case ENF_PUSHONE:
320                     printf("PUSHONE ");
321                     break;
322             #endif
323             #ifdef ENF_PUSHFFFF
324                 case ENF_PUSHFFFF:
325                     printf("PUSHFFFF ");
326                     break;
327             #endif
328             #ifdef ENF_PUSHFF00
329                 case ENF_PUSHFF00:
330                     printf("PUSHFF00 ");

```

```

331         break;
332     #endif
333     #ifdef ENF_PUSH00FF
334         case ENF_PUSH00FF:
335             printf("PUSH00FF ");
336             break;
337     #endif
338     case ENF_LOAD_OFFSET:
339         printf("LOAD_OFFSET ");
340         break;
341     case ENF_BRTR:
342         printf("BRTR ");
343         break;
344     case ENF_BRFL:
345         printf("BRFL ");
346         break;
347     case ENF_POP:
348         printf("POP ");
349         break;
350 }

352 if (action >= ENF_PUSHWORD)
353     printf("PUSHWORD %d ", action - ENF_PUSHWORD);

355 switch (op) {
356     case ENF_EQ:
357         printf("EQ ");
358         break;
359     case ENF_LT:
360         printf("LT ");
361         break;
362     case ENF_LE:
363         printf("LE ");
364         break;
365     case ENF_GT:
366         printf("GT ");
367         break;
368     case ENF_GE:
369         printf("GE ");
370         break;
371     case ENF_AND:
372         printf("AND ");
373         break;
374     case ENF_OR:
375         printf("OR ");
376         break;
377     case ENF_XOR:
378         printf("XOR ");
379         break;
380     case ENF_COR:
381         printf("COR ");
382         break;
383     case ENF_CAND:
384         printf("CAND ");
385         break;
386     case ENF_CNOR:
387         printf("CNOR ");
388         break;
389     case ENF_CNAND:
390         printf("CNAND ");
391         break;
392     case ENF_NEQ:
393         printf("NEQ ");
394         break;
395 }

```

```

397         if (action == ENF_PUSHLIT ||
398             action == ENF_LOAD_OFFSET ||
399             action == ENF_BRTR ||
400             action == ENF_BRFL) {
401             pc++;
402             printf("\n\t%3d:  %d (0x%04x)", pc - code, *pc, *pc);
403         }

405     printf("\n");
406 }
407 }

409 /*
410  * Emit packet filter code to check a
411  * field in the packet for a particular value.
412  * Need different code for each field size.
413  * Since the pf can only compare 16 bit quantities
414  * we have to use masking to compare byte values.
415  * Long word (32 bit) quantities have to be done
416  * as two 16 bit comparisons.
417  */
418 static void
419 pf_compare_value(int offset, uint_t len, uint_t val)
420 {
421     /*
422      * If the property being filtered on is absent in the media
423      * packet, error out.
424      */
425     if (offset == -1)
426         pr_err("filter option unsupported on media");

428     switch (len) {
429     case 1:
430         pf_emit(ENF_PUSHWORD + offset / 2);
431         #if defined(_BIG_ENDIAN)
432             if (offset % 2)
433         #else
434             if (!(offset % 2))
435         #endif
436         {
437             #ifdef ENF_PUSH00FF
438                 pf_emit(ENF_PUSH00FF | ENF_AND);
439             #else
440                 pf_emit(ENF_PUSHLIT | ENF_AND);
441                 pf_emit(0x00FF);
442             #endif
443             pf_emit(ENF_PUSHLIT | ENF_EQ);
444             pf_emit(val);
445         } else {
446             #ifdef ENF_PUSHFF00
447                 pf_emit(ENF_PUSHFF00 | ENF_AND);
448             #else
449                 pf_emit(ENF_PUSHLIT | ENF_AND);
450                 pf_emit(0xFF00);
451             #endif
452             pf_emit(ENF_PUSHLIT | ENF_EQ);
453             pf_emit(val << 8);
454         }
455         break;

457     case 2:
458         pf_emit(ENF_PUSHWORD + offset / 2);
459         pf_emit(ENF_PUSHLIT | ENF_EQ);
460         pf_emit((ushort_t)val);
461         break;

```

```

463     case 4:
464         pf_emit(ENF_PUSHWORD + offset / 2);
465         pf_emit(ENF_PUSHLIT | ENF_EQ);
466         #if defined(_BIG_ENDIAN)
467             pf_emit(val >> 16);
468         #elif defined(_LITTLE_ENDIAN)
469             pf_emit(val & 0xffff);
470         #else
471             #error One of _BIG_ENDIAN and _LITTLE_ENDIAN must be defined
472         #endif
473         pf_emit(ENF_PUSHWORD + (offset / 2) + 1);
474         pf_emit(ENF_PUSHLIT | ENF_EQ);
475         #if defined(_BIG_ENDIAN)
476             pf_emit(val & 0xffff);
477         #else
478             pf_emit(val >> 16);
479         #endif
480         pf_emit(ENF_AND);
481         break;
482     }
483 }

485 /*
486  * same as pf_compare_value, but only for emitting code to
487  * compare ipv6 addresses.
488  */
489 static void
490 pf_compare_value_v6(int offset, uint_t len, struct in6_addr val)
491 {
492     int i;

494     for (i = 0; i < len; i += 2) {
495         pf_emit(ENF_PUSHWORD + offset / 2 + i / 2);
496         pf_emit(ENF_PUSHLIT | ENF_EQ);
497         pf_emit(*(uint16_t *)&val.s6_addr[i]);
498         if (i != 0)
499             pf_emit(ENF_AND);
500     }
501 }

504 /*
505  * Same as above except mask the field value
506  * before doing the comparison. The comparison checks
507  * to make sure the values are equal.
508  */
509 static void
510 pf_compare_value_mask(int offset, uint_t len, uint_t val, int mask)
511 {
512     pf_compare_value_mask_generic(offset, len, val, mask, ENF_EQ);
513 }

515 /*
516  * Same as above except the values are compared to see if they are not
517  * equal.
518  */
519 static void
520 pf_compare_value_mask_neq(int offset, uint_t len, uint_t val, int mask)
521 {
522     pf_compare_value_mask_generic(offset, len, val, mask, ENF_NEQ);
523 }

525 /*
526  * Similar to pf_compare_value.
527  * This is the utility function that does the actual work to compare

```



```

529 * two values using a mask. The comparison operation is passed into
530 * the function.
531 */
532 static void
533 pf_compare_value_mask_generic(int offset, uint_t len, uint_t val, int mask,
534                               uint_t op)
535 {
536     /*
537      * If the property being filtered on is absent in the media
538      * packet, error out.
539      */
540     if (offset == -1)
541         pr_err("filter option unsupported on media");
542
543     switch (len) {
544     case 1:
545         pf_emit(ENF_PUSHPWORD + offset / 2);
546 #if defined(_BIG_ENDIAN)
547         if (offset % 2)
548 #else
549         if (!offset % 2)
550 #endif
551         {
552             pf_emit(ENF_PUSHLIT | ENF_AND);
553             pf_emit(mask & 0x00ff);
554             pf_emit(ENF_PUSHLIT | op);
555             pf_emit(val);
556         } else {
557             pf_emit(ENF_PUSHLIT | ENF_AND);
558             pf_emit((mask << 8) & 0xff00);
559             pf_emit(ENF_PUSHLIT | op);
560             pf_emit(val << 8);
561         }
562         break;
563
564     case 2:
565         pf_emit(ENF_PUSHPWORD + offset / 2);
566         pf_emit(ENF_PUSHLIT | ENF_AND);
567         pf_emit(htons((ushort_t)mask));
568         pf_emit(ENF_PUSHLIT | op);
569         pf_emit(htons((ushort_t)val));
570         break;
571
572     case 4:
573         pf_emit(ENF_PUSHPWORD + offset / 2);
574         pf_emit(ENF_PUSHLIT | ENF_AND);
575         pf_emit(htons((ushort_t)((mask >> 16) & 0xffff)));
576         pf_emit(ENF_PUSHLIT | op);
577         pf_emit(htons((ushort_t)((val >> 16) & 0xffff)));
578
579         pf_emit(ENF_PUSHPWORD + (offset / 2) + 1);
580         pf_emit(ENF_PUSHLIT | ENF_AND);
581         pf_emit(htons((ushort_t)(mask & 0xffff)));
582         pf_emit(ENF_PUSHLIT | op);
583         pf_emit(htons((ushort_t)(val & 0xffff)));
584
585         pf_emit(ENF_AND);
586         break;
587     }
588 }
589
590 /*
591 * Like pf_compare_value() but compare on a 32-bit zoneid value.
592 * The argument val passed in is in network byte order.
593 */
594 static void

```

```

595 pf_compare_zoneid(int offset, uint32_t val)
596 {
597     int i;
598
599     for (i = 0; i < sizeof(uint32_t) / 2; i++) {
600         pf_emit(ENF_PUSHPWORD + offset / 2 + i);
601         pf_emit(ENF_PUSHLIT | ENF_EQ);
602         pf_emit(((uint16_t *)&val)[i]);
603         if (i != 0)
604             pf_emit(ENF_AND);
605     }
606 }
607
608 /*
609 * Generate pf code to match an IPv4 or IPv6 address.
610 */
611 static void
612 pf_ipaddr_match(which, hostname, inet_type)
613     enum direction which;
614     char *hostname;
615     int inet_type;
616 {
617     bool_t found_host;
618     uint_t *addr4ptr;
619     uint_t addr4;
620     struct in6_addr *addr6ptr;
621     int h_addr_index;
622     struct hostent *hp = NULL;
623     int error_num = 0;
624     boolean_t first = B_TRUE;
625     int pass = 0;
626     int i;
627
628     /*
629      * The addr4offset and addr6offset variables simplify the code which
630      * generates the address comparison filter. With these two variables,
631      * duplicate code need not exist for the TO and FROM case.
632      * A value of -1 describes the ANY case (TO and FROM).
633      */
634     int addr4offset;
635     int addr6offset;
636
637     found_host = 0;
638
639     if (tokentype == ADDR_IP) {
640         hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
641         if (hp == NULL) {
642             if (error_num == TRY_AGAIN) {
643                 pr_err("could not resolve %s (try again later)",
644                     hostname);
645             } else {
646                 pr_err("could not resolve %s", hostname);
647             }
648         }
649         inet_type = IPV4_ONLY;
650     } else if (tokentype == ADDR_IP6) {
651         hp = getipnodebyname(hostname, AF_INET6, 0, &error_num);
652         if (hp == NULL) {
653             if (error_num == TRY_AGAIN) {
654                 pr_err("could not resolve %s (try again later)",
655                     hostname);
656             } else {
657                 pr_err("could not resolve %s", hostname);
658             }
659         }
660         inet_type = IPV6_ONLY;

```

```

661     } else if (tokentype == ALPHA) {
662         /* Some hostname i.e. tokentype is ALPHA */
663         switch (inet_type) {
664             case IPV4_ONLY:
665                 /* Only IPv4 address is needed */
666                 hp = getipnodebyname(hostname, AF_INET, 0, &error_num);
667                 if (hp != NULL) {
668                     found_host = 1;
669                 }
670                 break;
671             case IPV6_ONLY:
672                 /* Only IPv6 address is needed */
673                 hp = getipnodebyname(hostname, AF_INET6, 0, &error_num);
674                 if (hp != NULL) {
675                     found_host = 1;
676                 }
677                 break;
678             case IPV4_AND_IPV6:
679                 /* Both IPv4 and IPv6 are needed */
680                 hp = getipnodebyname(hostname, AF_INET6,
681                                     AI_ALL | AI_V4MAPPED, &error_num);
682                 if (hp != NULL) {
683                     found_host = 1;
684                 }
685                 break;
686             default:
687                 found_host = 0;
688         }
689
690         if (!found_host) {
691             if (error_num == TRY_AGAIN) {
692                 pr_err("could not resolve %s (try again later)",
693                     hostname);
694             } else {
695                 pr_err("could not resolve %s", hostname);
696             }
697         }
698     } else {
699         pr_err("unknown token type: %s", hostname);
700     }
701
702     switch (which) {
703     case TO:
704         addr4offset = IPV4_DSTADDR_OFFSET;
705         addr6offset = IPV6_DSTADDR_OFFSET;
706         break;
707     case FROM:
708         addr4offset = IPV4_SRCADDR_OFFSET;
709         addr6offset = IPV6_SRCADDR_OFFSET;
710         break;
711     case ANY:
712         addr4offset = -1;
713         addr6offset = -1;
714         break;
715     }
716
717     if (hp != NULL && hp->h_addrtype == AF_INET) {
718         pf_matchfn("ip");
719         if (dl.dl_type == DL_ETHER)
720             pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
721         h_addr_index = 0;
722         addr4ptr = (uint_t *)hp->h_addr_list[h_addr_index];
723         while (addr4ptr != NULL) {
724             if (addr4offset == -1) {
725                 pf_compare_value(IPV4_SRCADDR_OFFSET, 4,
726                     *addr4ptr);

```

```

727         if (h_addr_index != 0)
728             pf_emit(ENF_OR);
729         pf_compare_value(IPV4_DSTADDR_OFFSET, 4,
730             *addr4ptr);
731         pf_emit(ENF_OR);
732     } else {
733         pf_compare_value(addr4offset, 4,
734             *addr4ptr);
735         if (h_addr_index != 0)
736             pf_emit(ENF_OR);
737     }
738     addr4ptr = (uint_t *)hp->h_addr_list[++h_addr_index];
739     pf_emit(ENF_AND);
740 } else {
741     /* first pass: IPv4 addresses */
742     h_addr_index = 0;
743     addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
744     first = B_TRUE;
745     while (addr6ptr != NULL) {
746         if (IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
747             if (first) {
748                 pf_matchfn("ip");
749                 if (dl.dl_type == DL_ETHER) {
750                     pf_check_vlan_tag(
751                         ENCAP_ETHERTYPE_OFF/2);
752                 }
753                 pass++;
754             }
755             IN6_V4MAPPED_TO_INADDR(addr6ptr,
756                 (struct in_addr *)&addr4);
757             if (addr4offset == -1) {
758                 pf_compare_value(IPV4_SRCADDR_OFFSET, 4,
759                     addr4);
760                 if (!first)
761                     pf_emit(ENF_OR);
762                 pf_compare_value(IPV4_DSTADDR_OFFSET, 4,
763                     addr4);
764                 pf_emit(ENF_OR);
765             } else {
766                 pf_compare_value(addr4offset, 4,
767                     addr4);
768                 if (!first)
769                     pf_emit(ENF_OR);
770             }
771             if (first)
772                 first = B_FALSE;
773         }
774         addr6ptr = (struct in6_addr *)
775             hp->h_addr_list[++h_addr_index];
776     }
777     if (!first) {
778         pf_emit(ENF_AND);
779     }
780     /* second pass: IPv6 addresses */
781     h_addr_index = 0;
782     addr6ptr = (struct in6_addr *)hp->h_addr_list[h_addr_index];
783     first = B_TRUE;
784     while (addr6ptr != NULL) {
785         if (!IN6_IS_ADDR_V4MAPPED(addr6ptr)) {
786             if (first) {
787                 pf_matchfn("ip6");
788                 if (dl.dl_type == DL_ETHER) {
789                     pf_check_vlan_tag(
790                         ENCAP_ETHERTYPE_OFF/2);
791                 }

```

```

793         pass++;
794     }
795     if (addr6offset == -1) {
796         pf_compare_value_v6(IPV6_SRCADDR_OFFSET,
797             16, *addr6ptr);
798         if (!first)
799             pf_emit(ENF_OR);
800         pf_compare_value_v6(IPV6_DSTADDR_OFFSET,
801             16, *addr6ptr);
802         pf_emit(ENF_OR);
803     } else {
804         pf_compare_value_v6(addr6offset, 16,
805             *addr6ptr);
806         if (!first)
807             pf_emit(ENF_OR);
808     }
809     if (first)
810         first = B_FALSE;
811 }
812 addr6ptr = (struct in6_addr *)
813 hp->h_addr_list[++h_addr_index];
814 }
815 if (!first) {
816     pf_emit(ENF_AND);
817 }
818 if (pass == 2) {
819     pf_emit(ENF_OR);
820 }
821 }
822
823 if (hp != NULL) {
824     freehostent(hp);
825 }
826 }

```

```

829 static void
830 pf_compare_address(int offset, uint_t len, uchar_t *addr)
831 {
832     uint32_t val;
833     uint16_t sval;
834     boolean_t didone = B_FALSE;
835
836     /*
837      * If the property being filtered on is absent in the media
838      * packet, error out.
839      */
840     if (offset == -1)
841         pr_err("filter option unsupported on media");
842
843     while (len > 0) {
844         if (len >= 4) {
845             (void) memcpy(&val, addr, 4);
846             pf_compare_value(offset, 4, val);
847             addr += 4;
848             offset += 4;
849             len -= 4;
850         } else if (len >= 2) {
851             (void) memcpy(&sval, addr, 2);
852             pf_compare_value(offset, 2, sval);
853             addr += 2;
854             offset += 2;
855             len -= 2;
856         } else {
857             pf_compare_value(offset++, 1, *addr++);
858             len--;
859         }
860     }
861 }

```

```

859     }
860     if (didone)
861         pf_emit(ENF_AND);
862     didone = B_TRUE;
863 }
864 }
865
866 /*
867  * Compare ethernet addresses.
868  */
869 static void
870 pf_etheraddr_match(which, hostname)
871     enum direction which;
872     char *hostname;
873 {
874     struct ether_addr e, *ep = NULL;
875
876     if (isxdigit(*hostname))
877         ep = ether_aton(hostname);
878     if (ep == NULL) {
879         if (ether_hostton(hostname, &e))
880             if (!arp_for_ether(hostname, &e))
881                 pr_err("cannot obtain ether addr for %s",
882                     hostname);
883         ep = &e;
884     }
885
886     pf_clear_offset_register();
887
888     switch (which) {
889     case TO:
890         pf_compare_address(dl.dl_link_dest_offset, dl.dl_link_addr_len,
891             (uchar_t *)ep);
892         break;
893     case FROM:
894         pf_compare_address(dl.dl_link_src_offset, dl.dl_link_addr_len,
895             (uchar_t *)ep);
896         break;
897     case ANY:
898         pf_compare_address(dl.dl_link_dest_offset, dl.dl_link_addr_len,
899             (uchar_t *)ep);
900         pf_compare_address(dl.dl_link_src_offset, dl.dl_link_addr_len,
901             (uchar_t *)ep);
902         pf_emit(ENF_OR);
903         break;
904     }
905 }
906
907 /*
908  * Emit code to compare the network part of
909  * an IP address.
910  */
911 static void
912 pf_netaddr_match(which, netname)
913     enum direction which;
914     char *netname;
915 {
916     uint_t addr;
917     uint_t mask = 0xff000000;
918     struct netent *np;
919
920     if (isdigit(*netname)) {
921         addr = inet_network(netname);
922     } else {
923         np = getnetbyname(netname);
924         if (np == NULL)

```

```

925         pr_err("net %s not known", netname);
926         addr = np->n_net;
927     }

929     /*
930     * Left justify the address and figure
931     * out a mask based on the supplied address.
932     * Set the mask according to the number of zero
933     * low-order bytes.
934     * Note: this works only for whole octet masks.
935     */
936     if (addr) {
937         while ((addr & ~mask) != 0) {
938             mask |= (mask >> 8);
939         }
940     }

942     pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);

944     switch (which) {
945     case TO:
946         pf_compare_value_mask(IPV4_DSTADDR_OFFSET, 4, addr, mask);
947         break;
948     case FROM:
949         pf_compare_value_mask(IPV4_SRCADDR_OFFSET, 4, addr, mask);
950         break;
951     case ANY:
952         pf_compare_value_mask(IPV4_SRCADDR_OFFSET, 4, addr, mask);
953         pf_compare_value_mask(IPV4_DSTADDR_OFFSET, 4, addr, mask);
954         pf_emit(ENF_OR);
955         break;
956     }
957 }

959 /*
960 * Emit code to match on src or destination zoneid.
961 * The zoneid passed in is in network byte order.
962 */
963 static void
964 pf_match_zone(enum direction which, uint32_t zoneid)
965 {
966     if (dl.dl_type != DL_IPNET)
967         pr_err("zone filter option unsupported on media");

969     switch (which) {
970     case TO:
971         pf_compare_zoneid(IPNET_DSTZONE_OFFSET, zoneid);
972         break;
973     case FROM:
974         pf_compare_zoneid(IPNET_SRCZONE_OFFSET, zoneid);
975         break;
976     case ANY:
977         pf_compare_zoneid(IPNET_SRCZONE_OFFSET, zoneid);
978         pf_compare_zoneid(IPNET_DSTZONE_OFFSET, zoneid);
979         pf_emit(ENF_OR);
980         break;
981     }
982 }

984 /*
985 * A helper function to keep the code to emit instructions
986 * to change the offset register in one place.
987 *
988 * INPUTS: offset - An value representing an offset in 16-bit
989 *          words.
990 * OUTPUTS: If there is enough room in the storage for the

```

```

991 *          packet filtering program, instructions to load
992 *          a constant to the offset register. Otherwise,
993 *          nothing.
994 */
995 static void
996 pf_emit_load_offset(uint_t offset)
997 {
998     pf_emit(ENF_LOAD_OFFSET | ENF_NOP);
999     pf_emit(offset);
1000 }

1002 /*
1003 * Clear pfmod's offset register.
1004 */
1005 * INPUTS: none
1006 * OUTPUTS: Instructions to clear the offset register if
1007 *           there is enough space remaining in the packet
1008 *           filtering program structure's storage, and
1009 *           the last thing done to the offset register was
1010 *           not clearing the offset register. Otherwise,
1011 *           nothing.
1012 */
1013 static void
1014 pf_clear_offset_register()
1015 {
1016     if (last_offset_operation != (void*)pf_clear_offset_register) {
1017         pf_emit_load_offset(0);
1018         last_offset_operation = (void*)pf_clear_offset_register;
1019     }
1020 }

1022 /*
1023 * This function will issue opcodes to check if a packet
1024 * is VLAN tagged, and if so, update the offset register
1025 * with the appropriate offset.
1026 *
1027 * Note that if the packet is not VLAN tagged, then the offset
1028 * register will be cleared.
1029 *
1030 * If the interface type is not an ethernet type, then this
1031 * function returns without doing anything.
1032 *
1033 * If the last attempt to change the offset register occurred because
1034 * of a call to this function that was called with the same offset,
1035 * then we don't issue packet filtering instructions.
1036 *
1037 * INPUTS: offset - an offset in 16 bit words. The function
1038 *           will set the offset register to this
1039 *           value if the packet is VLAN tagged.
1040 * OUTPUTS: If the conditions are met, packet filtering instructions.
1041 */
1042 static void
1043 pf_check_vlan_tag(uint_t offset)
1044 {
1045     static uint_t last_offset = 0;

1047     if ((interface->mac_type == DL_ETHER ||
1048         interface->mac_type == DL_CSMACD) &&
1049         (last_offset_operation != (void*)pf_check_vlan_tag ||
1050          last_offset != offset)) {
1051         /*
1052         * First thing is to clear the offset register.
1053         * We don't know what state it is in, and if it
1054         * is not zero, then we have no idea what we load
1055         * when we execute ENF_PUSHPWORD.
1056         */

```

```

1057     pf_clear_offset_register();
1059     /*
1060      * Check the ethertype.
1061      */
1062     pf_compare_value(dl.dl_link_type_offset, 2,
1063                     htons(ETHERTYPE_VLAN));
1065     /*
1066      * And if it's not VLAN, don't load offset to the offset
1067      * register.
1068      */
1069     pf_emit(ENF_BRFL | ENF_NOP);
1070     pf_emit(3);
1072     /*
1073      * Otherwise, load offset to the offset register.
1074      */
1075     pf_emit_load_offset(offset);
1077     /*
1078      * Now get rid of the results of the comparison,
1079      * we don't want the results of the comparison to affect
1080      * other logic in the packet filtering program.
1081      */
1082     pf_emit(ENF_POP | ENF_NOP);
1084     /*
1085      * Set the last operation at the end, or any time
1086      * after the call to pf_clear_offset because
1087      * pf_clear_offset uses it.
1088      */
1089     last_offset_operation = (void*)pf_check_vlan_tag;
1090     last_offset = offset;
1091 }
1092 }
1094 /*
1095  * Utility function used to emit packet filtering code
1096  * to match an ethertype.
1097  *
1098  * INPUTS:  ethertype - The ethertype we want to check for.
1099  *          Don't call htons on the ethertype before
1100  *          calling this function.
1101  * OUTPUTS: If there is sufficient storage available, packet
1102  *          filtering code to check an ethertype. Otherwise,
1103  *          nothing.
1104  */
1105 static void
1106 pf_match_ethertype(uint_t ethertype)
1107 {
1108     /*
1109      * If the user wants to filter on ethertype VLAN,
1110      * then clear the offset register so that the offset
1111      * for ENF_PUSHPWORD points to the right place in the
1112      * packet.
1113      *
1114      * Otherwise, call pf_check_vlan_tag to set the offset
1115      * register such that the contents of the offset register
1116      * plus the argument for ENF_PUSHPWORD point to the right
1117      * part of the packet, whether or not the packet is VLAN
1118      * tagged. We call pf_check_vlan_tag with an offset of
1119      * two words because if the packet is VLAN tagged, we have
1120      * to move past the ethertype in the ethernet header, and
1121      * past the lower two octets of the VLAN header to get to
1122      * the ethertype in the VLAN header.

```

```

1123     */
1124     if (ethertype == ETHERTYPE_VLAN)
1125         pf_clear_offset_register();
1126     else
1127         pf_check_vlan_tag(2);
1129     pf_compare_value(dl.dl_link_type_offset, 2, htons(ethertype));
1130 }
1132 static void
1133 pf_match_ipnettype(uint_t type)
1134 {
1135     pf_compare_value(dl.dl_link_type_offset, 2, htons(type));
1136 }
1138 static void
1139 pf_match_ibtype(uint_t type)
1140 {
1141     pf_compare_value(dl.dl_link_type_offset, 2, htons(type));
1142 }
1144 /*
1145  * This function uses the table above to generate a
1146  * piece of a packet filtering program to check a transport
1147  * protocol type.
1148  *
1149  * INPUTS:  tranport_protocol - the transport protocol we're
1150  *          interested in.
1151  * OUTPUTS: If there is sufficient storage, then packet filtering
1152  *          code to check a transport protocol type. Otherwise,
1153  *          nothing.
1154  */
1155 static void
1156 pf_check_transport_protocol(uint_t transport_protocol)
1157 {
1158     int i;
1159     uint_t number_of_matches = 0;
1161     for (i = 0; dl.dl_trans_map_tbl[i].transport_protocol != -1; i++) {
1162         if (transport_protocol ==
1163             (uint_t)dl.dl_trans_map_tbl[i].transport_protocol) {
1164             number_of_matches++;
1165             dl.dl_match_fn(dl.dl_trans_map_tbl[i].network_protocol);
1166             pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
1167             pf_compare_value(dl.dl_trans_map_tbl[i].offset +
1168                             dl.dl_link_header_len, 1,
1169                             transport_protocol);
1170             pf_emit(ENF_AND);
1171             if (number_of_matches > 1) {
1172                 /*
1173                  * Since we have two or more matches, in
1174                  * order to have a correct and complete
1175                  * program we need to OR the result of
1176                  * each block of comparisons together.
1177                  */
1178                 pf_emit(ENF_OR);
1179             }
1180         }
1181     }
1182 }
1184 static void
1185 pf_matchfn(const char *proto)
1186 {
1187     int i;

```

```

1189     for (i = 0; dl.dl_net_map_tbl[i].nmt_val != -1; i++) {
1190         if (strcmp(proto, dl.dl_net_map_tbl[i].nmt_name) == 0) {
1191             dl.dl_match_fn(dl.dl_net_map_tbl[i].nmt_val);
1192             break;
1193         }
1194     }
1195 }

1197 static void
1198 pf_primary()
1199 {
1200     for (;;) {
1201         if (tokentype == FIELD)
1202             break;

1204         if (EQ("ip")) {
1205             pf_matchfn("ip");
1206             opstack++;
1207             next();
1208             break;
1209         }

1211         if (EQ("ip6")) {
1212             pf_matchfn("ip6");
1213             opstack++;
1214             next();
1215             break;
1216         }

1218         if (EQ("pppoe")) {
1219             pf_matchfn("pppoe");
1220             pf_match_ether_type(ETHERTYPE_PPPOES);
1221             pf_emit(ENF_OR);
1222             opstack++;
1223             next();
1224             break;
1225         }

1227         if (EQ("pppoed")) {
1228             pf_matchfn("pppoed");
1229             opstack++;
1230             next();
1231             break;
1232         }

1234         if (EQ("pppoe6")) {
1235             pf_matchfn("pppoe6");
1236             opstack++;
1237             next();
1238             break;
1239         }

1241         if (EQ("arp")) {
1242             pf_matchfn("arp");
1243             opstack++;
1244             next();
1245             break;
1246         }

1248         if (EQ("vlan")) {
1249             pf_matchfn("vlan");
1250             pf_compare_value_mask_neq(VLAN_ID_OFFSET, 2,
1251                                     0, VLAN_ID_MASK);
1252             pf_emit(ENF_AND);
1253             opstack++;
1254             next();

```

```

1255         break;
1256     }

1258     if (EQ("vlan-id")) {
1259         next();
1260         if (tokentype != NUMBER)
1261             pf_err("VLAN ID expected");
1262         pf_matchfn("vlan-id");
1263         pf_compare_value_mask(VLAN_ID_OFFSET, 2, tokenval,
1264                             VLAN_ID_MASK);
1265         pf_emit(ENF_AND);
1266         opstack++;
1267         next();
1268         break;
1269     }

1271     if (EQ("rarp")) {
1272         pf_matchfn("rarp");
1273         opstack++;
1274         next();
1275         break;
1276     }

1278     if (EQ("tcp")) {
1279         pf_check_transport_protocol(IPPROTO_TCP);
1280         opstack++;
1281         next();
1282         break;
1283     }

1285     if (EQ("udp")) {
1286         pf_check_transport_protocol(IPPROTO_UDP);
1287         opstack++;
1288         next();
1289         break;
1290     }

1292     if (EQ("ospf")) {
1293         pf_check_transport_protocol(IPPROTO_OSPF);
1294         opstack++;
1295         next();
1296         break;
1297     }

1300     if (EQ("sctp")) {
1301         pf_check_transport_protocol(IPPROTO_SCTP);
1302         opstack++;
1303         next();
1304         break;
1305     }

1307     if (EQ("icmp")) {
1308         pf_check_transport_protocol(IPPROTO_ICMP);
1309         opstack++;
1310         next();
1311         break;
1312     }

1314     if (EQ("icmp6")) {
1315         pf_check_transport_protocol(IPPROTO_ICMPV6);
1316         opstack++;
1317         next();
1318         break;
1319     }

```

```

1321     if (EQ("ip-in-ip")) {
1322         pf_check_transport_protocol(IPPROTO_ENCAP);
1323         opstack++;
1324         next();
1325         break;
1326     }
1327
1328     if (EQ("esp")) {
1329         pf_check_transport_protocol(IPPROTO_ESP);
1330         opstack++;
1331         next();
1332         break;
1333     }
1334
1335     if (EQ("ah")) {
1336         pf_check_transport_protocol(IPPROTO_AH);
1337         opstack++;
1338         next();
1339         break;
1340     }
1341
1342     if (EQ("dccp")) {
1343         pf_check_transport_protocol(IPPROTO_DCCP);
1344         opstack++;
1345         next();
1346         break;
1347     }
1348
1349 #endif /* ! codereview */
1350     if (EQ("(")) {
1351         inBrace++;
1352         next();
1353         pf_expression();
1354         if (EQ("(")) {
1355             if (inBrace)
1356                 inBraceOR--;
1357             inBrace--;
1358             next();
1359         }
1360         break;
1361     }
1362
1363     if (EQ("to") || EQ("dst")) {
1364         dir = TO;
1365         next();
1366         continue;
1367     }
1368
1369     if (EQ("from") || EQ("src")) {
1370         dir = FROM;
1371         next();
1372         continue;
1373     }
1374
1375     if (EQ("ether")) {
1376         eaddr = 1;
1377         next();
1378         continue;
1379     }
1380
1381     if (EQ("inet")) {
1382         next();
1383         if (EQ("host"))
1384             next();
1385         if (tokentype != ALPHA && tokentype != ADDR_IP)
1386             pr_err("host/IPv4 addr expected after inet");

```

```

1387         pf_ipaddr_match(dir, token, IPV4_ONLY);
1388         opstack++;
1389         next();
1390         break;
1391     }
1392
1393     if (EQ("inet6")) {
1394         next();
1395         if (EQ("host"))
1396             next();
1397         if (tokentype != ALPHA && tokentype != ADDR_IP6)
1398             pr_err("host/IPv6 addr expected after inet6");
1399         pf_ipaddr_match(dir, token, IPV6_ONLY);
1400         opstack++;
1401         next();
1402         break;
1403     }
1404
1405     if (EQ("proto")) {
1406         next();
1407         if (tokentype != NUMBER)
1408             pr_err("IP proto type expected");
1409         pf_check_vlan_tag(ENCAP_ETHERTYPE_OFF/2);
1410         pf_compare_value(
1411             IPV4_TYPE_HEADER_OFFSET + dl.dl_link_header_len, 1,
1412             tokenval);
1413         opstack++;
1414         next();
1415         break;
1416     }
1417
1418     if (EQ("broadcast")) {
1419         pf_clear_offset_register();
1420         pf_compare_value(dl.dl_link_dest_offset, 4, 0xffffffff);
1421         opstack++;
1422         next();
1423         break;
1424     }
1425
1426     if (EQ("multicast")) {
1427         pf_clear_offset_register();
1428         pf_compare_value_mask(
1429             dl.dl_link_dest_offset, 1, 0x01, 0x01);
1430         opstack++;
1431         next();
1432         break;
1433     }
1434
1435     if (EQ("ethertype")) {
1436         next();
1437         if (tokentype != NUMBER)
1438             pr_err("ether type expected");
1439         pf_match_ethertype(tokenval);
1440         opstack++;
1441         next();
1442         break;
1443     }
1444
1445     if (EQ("net") || EQ("dstnet") || EQ("srcnet")) {
1446         if (EQ("dstnet"))
1447             dir = TO;
1448         else if (EQ("srcnet"))
1449             dir = FROM;
1450         next();
1451         pf_netaddr_match(dir, token);
1452         dir = ANY;

```

```

1453         opstack++;
1454         next();
1455         break;
1456     }
1457
1458     if (EQ("zone")) {
1459         next();
1460         if (tokentype != NUMBER)
1461             pr_err("zoneid expected after inet");
1462         pf_match_zone(dir, BE_32((uint32_t)(tokenval)));
1463         opstack++;
1464         next();
1465         break;
1466     }
1467
1468     /*
1469     * Give up on anything that's obviously
1470     * not a primary.
1471     */
1472     if (EQ("and") || EQ("or") ||
1473         EQ("not") || EQ("decnet") || EQ("apple") ||
1474         EQ("length") || EQ("less") || EQ("greater") ||
1475         EQ("port") || EQ("srcport") || EQ("dstport") ||
1476         EQ("rpc") || EQ("gateway") || EQ("nofrag") ||
1477         EQ("bootp") || EQ("dhcp") || EQ("dhcp6") ||
1478         EQ("slp") || EQ("ldap")) {
1479         break;
1480     }
1481
1482     if (EQ("host") || EQ("between")) {
1483         tokentype == ALPHA || /* assume its a hostname */
1484         tokentype == ADDR_IP ||
1485         tokentype == ADDR_IP6 ||
1486         tokentype == ADDR_ETHER) {
1487         if (EQ("host") || EQ("between"))
1488             next();
1489         if (eaddr || tokentype == ADDR_ETHER) {
1490             pf_etheraddr_match(dir, token);
1491         } else if (tokentype == ALPHA) {
1492             pf_ipaddr_match(dir, token, IPV4_AND_IPV6);
1493         } else if (tokentype == ADDR_IP) {
1494             pf_ipaddr_match(dir, token, IPV4_ONLY);
1495         } else {
1496             pf_ipaddr_match(dir, token, IPV6_ONLY);
1497         }
1498         dir = ANY;
1499         eaddr = 0;
1500         opstack++;
1501         next();
1502         break;
1503     }
1504
1505     break; /* unknown token */
1506 }
1507
1508 static void
1509 pf_alteration()
1510 {
1511     int s = opstack;
1512
1513     pf_primary();
1514     for (;;) {
1515         if (EQ("and"))
1516             next();
1517         pf_primary();

```

```

1519         if (opstack != s + 2)
1520             break;
1521         pf_emit(ENF_AND);
1522         opstack--;
1523     }
1524 }
1525
1526 static void
1527 pf_expression()
1528 {
1529     pf_alteration();
1530     while (EQ("or") || EQ(",")) {
1531         if (inBrace)
1532             inBraceOR++;
1533         else
1534             foundOR++;
1535         next();
1536         pf_alteration();
1537         pf_emit(ENF_OR);
1538         opstack--;
1539     }
1540 }
1541
1542 /*
1543 * Attempt to compile the expression
1544 * in the string "e". If we can generate
1545 * pf code for it then return 1 - otherwise
1546 * return 0 and leave it up to the user-level
1547 * filter.
1548 */
1549 int
1550 pf_compile(e, print)
1551     char *e;
1552     int print;
1553 {
1554     char *argstr;
1555     char *sav_str, *ptr, *sav_ptr;
1556     int inBr = 0, aheadOR = 0;
1557
1558     argstr = strdup(e);
1559     sav_str = e;
1560     tkp = argstr;
1561     dir = ANY;
1562
1563     pfp = &pf.Pf_Filter[0];
1564     if (setjmp(env)) {
1565         return (0);
1566     }
1567
1568     /*
1569     * Set media specific packet offsets that this code uses.
1570     */
1571     if (interface->mac_type == DL_ETHER) {
1572         dl.dl_type = DL_ETHER;
1573         dl.dl_match_fn = pf_match_ether_type;
1574         dl.dl_trans_map_tbl = ether_transport_mapping_table;
1575         dl.dl_net_map_tbl = ether_network_mapping_table;
1576         dl.dl_link_header_len = 14;
1577         dl.dl_link_type_offset = 12;
1578         dl.dl_link_dest_offset = 0;
1579         dl.dl_link_src_offset = 6;
1580         dl.dl_link_addr_len = 6;
1581     }
1582
1583     if (interface->mac_type == DL_IB) {
1584         dl.dl_type = DL_IB;

```



```

1585     dl.dl_link_header_len = 4;
1586     dl.dl_link_type_offset = 0;
1587     dl.dl_link_dest_offset = dl.dl_link_src_offset = -1;
1588     dl.dl_link_addr_len = 20;
1589     dl.dl_match_fn = pf_match_ibtype;
1590     dl.dl_trans_map_tbl = ib_transport_mapping_table;
1591     dl.dl_net_map_tbl = ib_network_mapping_table;
1592 }

1594 if (interface->mac_type == DL_IPNET) {
1595     dl.dl_type = DL_IPNET;
1596     dl.dl_link_header_len = 24;
1597     dl.dl_link_type_offset = 0;
1598     dl.dl_link_dest_offset = dl.dl_link_src_offset = -1;
1599     dl.dl_link_addr_len = -1;
1600     dl.dl_match_fn = pf_match_ipnettype;
1601     dl.dl_trans_map_tbl = ipnet_transport_mapping_table;
1602     dl.dl_net_map_tbl = ipnet_network_mapping_table;
1603 }

1605 next();
1606 pf_expression();

1608 if (tokentype != EOL) {
1609     /*
1610      * The idea here is to do as much filtering as possible in
1611      * the kernel. So even if we find a token we don't understand,
1612      * we try to see if we can still set up a portion of the filter
1613      * in the kernel and use the userland filter to filter the
1614      * remaining stuff. Obviously, if our filter expression is of
1615      * type A AND B, we can filter A in kernel and then apply B
1616      * to the packets that got through. The same is not true for
1617      * a filter of type A OR B. We can't apply A first and then B
1618      * on the packets filtered through A.
1619      *
1620      * (We need to keep track of the fact when we find an OR,
1621      * and the fact that we are inside brackets when we find OR.
1622      * The variable 'foundOR' tells us if there was an OR behind,
1623      * 'inBraceOR' tells us if we found an OR before we could find
1624      * the end brace i.e. ')', and variable 'aheadOR' checks if
1625      * there is an OR in the expression ahead. if either of these
1626      * cases become true, we can't split the filtering)
1627      */

1629     if (foundOR || inBraceOR) {
1630         /* FORGET IN KERNEL FILTERING */
1631         return (0);
1632     } else {
1633
1634         /* CHECK IF NO OR AHEAD */
1635         sav_ptr = (char *)((uintptr_t)sav_str +
1636                             (uintptr_t)sav_tkp -
1637                             (uintptr_t)argstr);
1638
1639         ptr = sav_ptr;
1640         while (*ptr != '\0') {
1641             switch (*ptr) {
1642                 case '(':
1643                     inBr++;
1644                     break;
1645                 case ')':
1646                     inBr--;
1647                     break;
1648                 case 'o':
1649                 case 'O':
1650                     if ((*ptr + 1) == 'R' ||
1651                         *(ptr + 1) == 'r') && !inBr)

```

```

1651         aheadOR = 1;
1652         break;
1653     case ',':
1654         if (!inBr)
1655             aheadOR = 1;
1656         break;
1657     }
1658     ptr++;
1659 }
1660 if (!aheadOR) {
1661     /* NO OR AHEAD, SPLIT UP THE FILTERING */
1662     pf.Pf_FilterLen = pfp - &pf.Pf_Filter[0];
1663     pf.Pf_Priority = 5;
1664     if (print) {
1665         pf_codeprint(&pf.Pf_Filter[0],
1666                     pf.Pf_FilterLen);
1667     }
1668     compile(sav_ptr, print);
1669     return (2);
1670 } else
1671     return (0);
1672 }
1673 }

1675 pf.Pf_FilterLen = pfp - &pf.Pf_Filter[0];
1676 pf.Pf_Priority = 5; /* unimportant, so long as > 2 */
1677 if (print) {
1678     pf_codeprint(&pf.Pf_Filter[0], pf.Pf_FilterLen);
1679 }
1680 return (1);
1681 }

```

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_rport.c

1

10831 Sat Aug 18 10:36:59 2012

new/usr/src/cmd/cmd-inet/usr.sbin/snoop/snoop_rport.c

dccp: options and features

_____unchanged_portion_omitted_____

```
140 char *
141 getportname(int proto, in_port_t port)
142 {
143     const struct porttable *p, *pt;
144
145     switch (proto) {
146     case IPPROTO_DCCP: /* fallthru */
147     case IPPROTO_SCTP: /* fallthru */
148     case IPPROTO_TCP: pt = pt_tcp; break;
149     case IPPROTO_UDP: pt = pt_udp; break;
150     default: return (NULL);
151     }
152
153     for (p = pt; p->pt_num; p++) {
154         if (port == p->pt_num)
155             return (p->pt_short);
156     }
157     return (NULL);
158 }
```

_____unchanged_portion_omitted_____

new/usr/src/cmd/devfsadm/misc_link.c

1

```
*****
18154 Sat Aug 18 10:37:00 2012
new/usr/src/cmd/devfsadm/misc_link.c
dccp: fix setsockopt bug
*****

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) 1998, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
24 */

26 #include <regex.h>
27 #include <devfsadm.h>
28 #include <stdio.h>
29 #include <strings.h>
30 #include <stdlib.h>
31 #include <limits.h>
32 #include <sys/zone.h>
33 #include <sys/zcons.h>
34 #include <sys/cpuid_drv.h>

36 static int display(di_minor_t minor, di_node_t node);
37 static int parallel(di_minor_t minor, di_node_t node);
38 static int node_slash_minor(di_minor_t minor, di_node_t node);
39 static int driver_minor(di_minor_t minor, di_node_t node);
40 static int node_name(di_minor_t minor, di_node_t node);
41 static int minor_name(di_minor_t minor, di_node_t node);
42 static int wifi_minor_name(di_minor_t minor, di_node_t node);
43 static int conskbd(di_minor_t minor, di_node_t node);
44 static int consms(di_minor_t minor, di_node_t node);
45 static int power_button(di_minor_t minor, di_node_t node);
46 static int fc_port(di_minor_t minor, di_node_t node);
47 static int printer_create(di_minor_t minor, di_node_t node);
48 static int se_hdlc_create(di_minor_t minor, di_node_t node);
49 static int ppm(di_minor_t minor, di_node_t node);
50 static int gpio(di_minor_t minor, di_node_t node);
51 static int av_create(di_minor_t minor, di_node_t node);
52 static int tsalarm_create(di_minor_t minor, di_node_t node);
53 static int ntwdt_create(di_minor_t minor, di_node_t node);
54 static int zcons_create(di_minor_t minor, di_node_t node);
55 static int cpuid(di_minor_t minor, di_node_t node);
56 static int glvc(di_minor_t minor, di_node_t node);
57 static int ses_callback(di_minor_t minor, di_node_t node);
58 static int kmdrv_create(di_minor_t minor, di_node_t node);

60 static devfsadm_create_t misc_cbt[] = {
61     { "pseudo", "ddi_pseudo", "(^sad$)",
```

new/usr/src/cmd/devfsadm/misc_link.c

2

```
62     TYPE_EXACT | DRV_RE, ILEVEL_0, node_slash_minor
63 },
64 { "pseudo", "ddi_pseudo", "zsh",
65     TYPE_EXACT | DRV_EXACT, ILEVEL_0, driver_minor
66 },
67 { "network", "ddi_network", NULL,
68     TYPE_EXACT, ILEVEL_0, minor_name
69 },
70 { "wifi", "ddi_network:wifi", NULL,
71     TYPE_EXACT, ILEVEL_0, wifi_minor_name
72 },
73 { "display", "ddi_display", NULL,
74     TYPE_EXACT, ILEVEL_0, display
75 },
76 { "parallel", "ddi_parallel", NULL,
77     TYPE_EXACT, ILEVEL_0, parallel
78 },
79 { "enclosure", DDI_NT SCSI_ENCLOSURE, NULL,
80     TYPE_EXACT, ILEVEL_0, ses_callback
81 },
82 { "pseudo", "ddi_pseudo", "(^winlock$)|(^pm$)",
83     TYPE_EXACT | DRV_RE, ILEVEL_0, node_name
84 },
85 { "pseudo", "ddi_pseudo", "conskbd",
86     TYPE_EXACT | DRV_EXACT, ILEVEL_0, conskbd
87 },
88 { "pseudo", "ddi_pseudo", "consms",
89     TYPE_EXACT | DRV_EXACT, ILEVEL_0, consms
90 },
91 { "pseudo", "ddi_pseudo", "rsm",
92     TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
93 },
94 { "pseudo", "ddi_pseudo",
95     "(^lockstat$)|(^SUNW,rtvc$)|(^vol$)|(^log$)|(^sy$)|"
96     "(^ksyms$)|(^clone$)|(^tl$)|(^tnf$)|(^kstat$)|(^mdesc$)|(^eprom$)|"
97     "(^ptsl$)|(^mm$)|(^wc$)|(^dump$)|(^cn$)|(^svvslo$)|(^ptm$)|"
98     "(^ptc$)|(^openeep$)|(^poll$)|(^sysmsg$)|(^random$)|(^trapstat$)|"
99     "(^cryptoadm$)|(^crypto$)|(^pool$)|(^poolctl$)|(^bl$)|(^kmdb$)|"
100     "(^sysevent$)|(^kssl$)|(^physmem$)",
101     TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name
102 },
103 { "pseudo", "ddi_pseudo",
104     "(^ip$)|(^tcp$)|(^udp$)|(^icmp$)|(^dccp$)|"
105     "(^ip6$)|(^tcp6$)|(^udp6$)|(^icmp6$)|(^dccp6$)|"
106     "(^ip$)|(^tcp$)|(^udp$)|(^icmp$)|"
107     "(^ip6$)|(^tcp6$)|(^udp6$)|(^icmp6$)|"
108     "(^rts$)|(^arp$)|(^ipsec$)|(^ipsecsp$)|(^keysock$)|(^spdsock$)|"
109     "(^nca$)|(^rds$)|(^sdp$)|(^ipnet$)|(^dipistub$)|(^bpf$)",
110     TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name
111 },
112 { "pseudo", "ddi_pseudo",
113     "(^ipf$)|(^ipnat$)|(^ipstate$)|(^ipauth$)|"
114     "(^ipsync$)|(^ipscan$)|(^iplookup$)",
115     TYPE_EXACT | DRV_RE, ILEVEL_0, minor_name,
116 },
117 { "pseudo", "ddi_pseudo", "dld",
118     TYPE_EXACT | DRV_EXACT, ILEVEL_0, node_name
119 },
120 { "pseudo", "ddi_pseudo",
121     "(^kdmouse$)|(^rootprop$)",
122     TYPE_EXACT | DRV_RE, ILEVEL_0, node_name
123 },
124 { "pseudo", "ddi_pseudo", "tod",
125     TYPE_EXACT | DRV_EXACT, ILEVEL_0, node_name
126 },
127 { "pseudo", "ddi_pseudo", "envctrl(two)?",
```

```

126     TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
127 },
128 { "pseudo", "ddi_pseudo", "fcode",
129   TYPE_EXACT | DRV_RE, ILEVEL_0, minor_name,
130 },
131 { "power_button", "ddi_power_button", NULL,
132   TYPE_EXACT, ILEVEL_0, power_button,
133 },
134 { "FC port", "ddi_ctl:devctl", "fp",
135   TYPE_EXACT | DRV_EXACT, ILEVEL_0, fc_port
136 },
137 { "printer", "ddi_printer", NULL,
138   TYPE_EXACT, ILEVEL_0, printer_create
139 },
140 { "pseudo", "ddi_pseudo", "se",
141   TYPE_EXACT | DRV_EXACT, ILEVEL_0, se_hdlc_create
142 },
143 { "ppm", "ddi_ppm", NULL,
144   TYPE_EXACT, ILEVEL_0, ppm
145 },
146 { "pseudo", "ddi_pseudo", "gpio_87317",
147   TYPE_EXACT | DRV_EXACT, ILEVEL_0, gpio
148 },
149 { "pseudo", "ddi_pseudo", "sckmdrv",
150   TYPE_EXACT | DRV_RE, ILEVEL_0, kmdrv_create,
151 },
152 { "pseudo", "ddi_pseudo", "oplkmdrv",
153   TYPE_EXACT | DRV_RE, ILEVEL_0, kmdrv_create,
154 },
155 { "av", "^ddi_av:(isoch|async)$", NULL,
156   TYPE_RE, ILEVEL_0, av_create,
157 },
158 { "pseudo", "ddi_pseudo", "tsalarm",
159   TYPE_EXACT | DRV_RE, ILEVEL_0, tsalarm_create,
160 },
161 { "pseudo", "ddi_pseudo", "ntwdt",
162   TYPE_EXACT | DRV_RE, ILEVEL_0, ntwdt_create,
163 },
164 { "pseudo", "ddi_pseudo", "daplt",
165   TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
166 },
167 { "pseudo", "ddi_pseudo", "zcons",
168   TYPE_EXACT | DRV_EXACT, ILEVEL_0, zcons_create,
169 },
170 { "pseudo", "ddi_pseudo", CPUID_DRIVER_NAME,
171   TYPE_EXACT | DRV_EXACT, ILEVEL_0, cpuid,
172 },
173 { "pseudo", "ddi_pseudo", "glvc",
174   TYPE_EXACT | DRV_EXACT, ILEVEL_0, glvc,
175 },
176 { "pseudo", "ddi_pseudo", "dm2s",
177   TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name,
178 },
179 { "pseudo", "ddi_pseudo", "nsmb",
180   TYPE_EXACT | DRV_EXACT, ILEVEL_1, minor_name,
181 },
182 { "pseudo", "ddi_pseudo", "mem_cache",
183   TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
184 },
185 { "pseudo", "ddi_pseudo", "fm",
186   TYPE_EXACT | DRV_RE, ILEVEL_1, minor_name,
187 },
188 { "pseudo", "ddi_pseudo", "smbsrv",
189   TYPE_EXACT | DRV_EXACT, ILEVEL_1, minor_name,
190 },
191 { "pseudo", "ddi_pseudo", "tpm",

```

```

192     TYPE_EXACT | DRV_EXACT, ILEVEL_0, minor_name
193   },
194 };
_____unchanged_portion_omitted_____

```

 128972 Sat Aug 18 10:37:00 2012

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

dccp: build fixes, mdb (vfs sonode missing)

_____unchanged_portion_omitted_____

```

3841 static const mdb_dcmd_t dcmds[] = {

3843     /* from genunix.c */
3844     { "as2proc", ":", "convert as to proc_t address", as2proc },
3845     { "binding_hash_entry", ":", "print driver names hash table entry",
3846       binding_hash_entry },
3847     { "callout", "?[-r|n] [-s|l] [-xhB] [-t | -ab nsec [-dkD]]"
3848       " [-C addr | -S seqid] [-f name|addr] [-p name| addr] [-T|L [-E]]"
3849       " [-FivA]",
3850       "display callouts", callout, callout_help },
3851     { "calloutid", "[-d|v] xid", "print callout by extended id",
3852       calloutid, calloutid_help },
3853     { "class", NULL, "print process scheduler classes", class },
3854     { "cpuinfo", "?[-v]", "print CPUs and runnable threads", cpuinfo },
3855     { "did2thread", "? kt_did", "find kernel thread for this id",
3856       did2thread },
3857     { "errorq", "?[-v]", "display kernel error queues", errorq },
3858     { "fd", ":[fd num]", "get a file pointer from an fd", fd },
3859     { "flipone", ":", "the vik_rev_level 2 special", flipone },
3860     { "lminfo", NULL, "print lock manager information", lminfo },
3861     { "ndi_event_hdl", "?", "print ndi_event_hdl", ndi_event_hdl },
3862     { "panicinfo", NULL, "print panic information", panicinfo },
3863     { "pid2proc", "?", "convert PID to proc_t address", pid2proc },
3864     { "project", NULL, "display kernel project(s)", project },
3865     { "ps", "[-fltztP]", "list processes (and associated thr,lwp)", ps },
3866     { "pgrep", "[-x] [-n | -o] pattern",
3867       "pattern match against all processes", pgrep },
3868     { "ptree", NULL, "print process tree", ptree },
3869     { "sysevent", "?[-sv]", "print sysevent pending or sent queue",
3870       sysevent },
3871     { "sysevent_channel", "?", "print sysevent channel database",
3872       sysevent_channel },
3873     { "sysevent_class_list", ":", "print sysevent class list",
3874       sysevent_class_list },
3875     { "sysevent_subclass_list", ":",
3876       "print sysevent subclass list", sysevent_subclass_list },
3877     { "system", NULL, "print contents of /etc/system file", sysfile },
3878     { "task", NULL, "display kernel task(s)", task },
3879     { "time", "[-dlx]", "display system time", time, time_help },
3880     { "vnode2path", ":[-F]", "vnode address to pathname", vnode2path },
3881     { "whereopen", ":", "given a vnode, dumps procs which have it open",
3882       whereopen },

3884     /* from bio.c */
3885     { "bufpagefind", ":[addr]", "find page_t on buf_t list", bufpagefind },

3887     /* from bitset.c */
3888     { "bitset", ":", "display a bitset", bitset, bitset_help },

3890     /* from contract.c */
3891     { "contract", "?", "display a contract", cmd_contract },
3892     { "ctevent", ":", "display a contract event", cmd_ctevent },
3893     { "ctid", ":", "convert id to a contract pointer", cmd_ctid },

3895     /* from cpupart.c */
3896     { "cpupart", "?[-v]", "print cpu partition info", cpupart },

3898     /* from cred.c */
3899     { "cred", ":[-v]", "display a credential", cmd_cred },

```

```

3900     { "credgrp", ":[-v]", "display cred_t groups", cmd_credgrp },
3901     { "credsid", ":[-v]", "display a credsid_t", cmd_creditsid },
3902     { "ksidlist", ":[-v]", "display a ksidlist_t", cmd_ksidlist },

3904     /* from cyclic.c */
3905     { "cyccover", NULL, "dump cyclic coverage information", cyccover },
3906     { "cycid", "?", "dump a cyclic id", cycid },
3907     { "cycinfo", "?", "dump cyc_cpu info", cycinfo },
3908     { "cyclic", ":", "developer information", cyclic },
3909     { "cyctrace", "?", "dump cyclic trace buffer", cyctrace },

3911     /* from damap.c */
3912     { "damap", ":", "display a damap_t", damap, damap_help },

3914     /* from devinfo.c */
3915     { "devbindings", "?[-qs] [device-name | major-num]",
3916       "print devinfo nodes bound to device-name or major-num",
3917       devbindings, devinfo_help },
3918     { "devinfo", ":[-qs]", "detailed devinfo of one node", devinfo,
3919       devinfo_help },
3920     { "devinfo_audit", ":[-v]", "devinfo configuration audit record",
3921       devinfo_audit },
3922     { "devinfo_audit_log", "?[-v]", "system wide devinfo configuration log",
3923       devinfo_audit_log },
3924     { "devinfo_audit_node", ":[-v]", "devinfo node configuration history",
3925       devinfo_audit_node },
3926     { "devinfo2driver", ":", "find driver name for this devinfo node",
3927       devinfo2driver },
3928     { "devnames", "?[-vm] [num]", "print devnames array", devnames },
3929     { "dev2major", "?<dev_t>", "convert dev_t to a major number",
3930       dev2major },
3931     { "dev2minor", "?<dev_t>", "convert dev_t to a minor number",
3932       dev2minor },
3933     { "devt", "?<dev_t>", "display a dev_t's major and minor numbers",
3934       devt },
3935     { "major2name", "?<major-num>", "convert major number to dev name",
3936       major2name },
3937     { "minornodes", ":", "given a devinfo node, print its minor nodes",
3938       minornodes },
3939     { "modctl2devinfo", ":", "given a modctl, list its devinfos",
3940       modctl2devinfo },
3941     { "name2major", "<dev-name>", "convert dev name to major number",
3942       name2major },
3943     { "prtconf", "?[-vpc]", "print devinfo tree", prtconf, prtconf_help },
3944     { "softstate", ":[instance]", "retrieve soft-state pointer",
3945       softstate },
3946     { "devinfo_fm", ":", "devinfo fault managment configuration",
3947       devinfo_fm },
3948     { "devinfo_fmce", ":", "devinfo fault managment cache entry",
3949       devinfo_fmce },

3951     /* from findstack.c */
3952     { "findstack", ":[-v]", "find kernel thread stack", findstack },
3953     { "findstack_debug", NULL, "toggle findstack debugging",
3954       findstack_debug },
3955     { "stacks", "?[-afiv] [-c func] [-C func] [-m module] [-M module] "
3956       "[-s subj | -S subj] [-t tstate | -T tstate]",
3957       "print unique kernel thread stacks",
3958       stacks, stacks_help },

3960     /* from fm.c */
3961     { "ereport", ":[-v]", "print ereports logged in dump",
3962       ereport },

3964     /* from group.c */
3965     { "group", "?[-g]", "display a group", group },

```

```

3967  /* from hotplug.c */
3968  { "hotplug", "?[-p]", "display a registered hotplug attachment",
3969    hotplug, hotplug_help },

3971  /* from irm.c */
3972  { "irmpools", NULL, "display interrupt pools", irmpools_dcmd },
3973  { "irmreqs", NULL, "display interrupt requests in an interrupt pool",
3974    irmreqs_dcmd },
3975  { "irmreq", NULL, "display an interrupt request", irmreq_dcmd },

3977  /* from kgrep.c + genunix.c */
3978  { "kgrep", KGREP_USAGE, "search kernel as for a pointer", kgrep,
3979    kgrep_help },

3981  /* from kmem.c */
3982  { "alloctb", ":", "given a thread, print its allocated buffers",
3983    alloctb },
3984  { "bufctl", ":[-vh] [-a addr] [-c caller] [-e earliest] [-l latest] "
3985    "[-t thd]", "print or filter a bufctl", bufctl, bufctl_help },
3986  { "freedby", ":", "given a thread, print its freed buffers", freedby },
3987  { "kmalog", "?[ fail | slab ]",
3988    "display kmem transaction log and stack traces", kmalog },
3989  { "kmastat", "[-kmg]", "kernel memory allocator stats",
3990    kmastat },
3991  { "kmausers", "?[-ef] [cache ...]", "current medium and large users "
3992    "of the kmem allocator", kmausers, kmausers_help },
3993  { "kmem_cache", "?[-n name]",
3994    "print kernel memory caches", kmem_cache, kmem_cache_help },
3995  { "kmem_slabs", "?[-v] [-n cache] [-N cache] [-b maxbins] "
3996    "[-B minbinsize]", "display slab usage per kmem cache",
3997    kmem_slabs, kmem_slabs_help },
3998  { "kmem_debug", NULL, "toggle kmem dcmd/walk debugging", kmem_debug },
3999  { "kmem_log", "?[-b]", "dump kmem transaction log", kmem_log },
4000  { "kmem_verify", "?", "check integrity of kmem-managed memory",
4001    kmem_verify },
4002  { "vmem", "?", "print a vmem_t", vmem },
4003  { "vmem_seg", ":[-sv] [-c caller] [-e earliest] [-l latest] "
4004    "[-m minsize] [-M maxsize] [-t thread] [-T type]",
4005    "print or filter a vmem_seg", vmem_seg, vmem_seg_help },
4006  { "whatthread", ":[-v]", "print threads whose stack contains the "
4007    "given address", whatthread },

4009  /* from ldi.c */
4010  { "ldi_handle", "?[-i]", "display a layered driver handle",
4011    ldi_handle, ldi_handle_help },
4012  { "ldi_ident", NULL, "display a layered driver identifier",
4013    ldi_ident, ldi_ident_help },

4015  /* from leaky.c + leaky_subr.c */
4016  { "findleaks", FINDLEAKS_USAGE,
4017    "search for potential kernel memory leaks", findleaks,
4018    findleaks_help },

4020  /* from lgrp.c */
4021  { "lgrp", "?[-q] [-p | -Pih]", "display an lgrp", lgrp },
4022  { "lgrp_set", "", "display bitmask of lgroups as a list", lgrp_set },

4024  /* from log.c */
4025  { "msgbuf", "?[-v]", "print most recent console messages", msgbuf },

4027  /* from mdi.c */
4028  { "mdipi", NULL, "given a path, dump mdi_pathinfo "
4029    "and detailed pi_prop list", mdipi },
4030  { "mdiprops", NULL, "given a pi_prop, dump the pi_prop list",
4031    mdiprops },

```

```

4032  { "mdiphci", NULL, "given a phci, dump mdi_phci and "
4033    "list all paths", mdiphci },
4034  { "mdivhci", NULL, "given a vhci, dump mdi_vhci and list "
4035    "all phcis", mdivhci },
4036  { "mdiclient_paths", NULL, "given a path, walk mdi_pathinfo "
4037    "client links", mdiclient_paths },
4038  { "mdiphci_paths", NULL, "given a path, walk through mdi_pathinfo "
4039    "phci links", mdiphci_paths },
4040  { "mdiphcis", NULL, "given a phci, walk through mdi_phci ph_next links",
4041    mdiphcis },

4043  /* from memory.c */
4044  { "addr2smap", ":[offset]", "translate address to smap", addr2smap },
4045  { "memlist", "?[-iav]", "display a struct memlist", memlist },
4046  { "memstat", NULL, "display memory usage summary", memstat },
4047  { "page", "?", "display a summarized page_t", page },
4048  { "pagelookup", "?[-v vp] [-o offset]",
4049    "find the page_t with the name {vp, offset}",
4050    pagelookup, pagelookup_help },
4051  { "page_num2pp", ":", "find the page_t for a given page frame number",
4052    page_num2pp },
4053  { "pmap", ":[-q]", "print process memory map", pmap },
4054  { "seg", ":", "print address space segment", seg },
4055  { "swapinfo", "?", "display a struct swapinfo", swapinfo },
4056  { "vnode2smap", ":[offset]", "translate vnode to smap", vnode2smap },

4058  /* from mmd.c */
4059  { "multidata", ":[-sv]", "display a summarized multidata_t",
4060    multidata },
4061  { "patttbl", ":", "display a summarized multidata attribute table",
4062    patttbl },
4063  { "pattr2multidata", ":", "print multidata pointer from pattr_t",
4064    pattr2multidata },
4065  { "pdesc2slab", ":", "print pdesc slab pointer from pdesc_t",
4066    pdesc2slab },
4067  { "pdesc_verify", ":", "verify integrity of a pdesc_t", pdesc_verify },
4068  { "slab2multidata", ":", "print multidata pointer from pdesc_slab_t",
4069    slab2multidata },

4071  /* from modhash.c */
4072  { "modhash", "?[-ceht] [-k key] [-v val] [-i index]",
4073    "display information about one or all mod_hash structures",
4074    modhash, modhash_help },
4075  { "modent", ":[-k | -v | -t type]",
4076    "display information about a mod_hash_entry", modent,
4077    modent_help },

4079  /* from net.c */
4080  { "dladm", "<sub-command> [flags]", "show data link information",
4081    dladm, dladm_help },
4082  { "mi", ":[-p] [-d | -m]", "filter and display MI object or payload",
4083    mi },
4084  { "netstat", "[-arv] [-f inet | inet6 | unix] [-P tcp | udp | icmp | dcc
4084    "netstat", "[-arv] [-f inet | inet6 | unix] [-P tcp | udp | icmp]",
4085    "show network statistics", netstat },
4086  { "sonode", "?[-f inet | inet6 | unix | #] "
4087    "[-t stream | dgram | raw | #] [-p #]",
4088    "filter and display sonode", sonode },

4090  /* from netstack.c */
4091  { "netstack", "", "show stack instances", netstack },

4093  /* from nvpair.c */
4094  { NVPAIR_DCMD_NAME, NVPAIR_DCMD_USAGE, NVPAIR_DCMD_DESCR,
4095    nvpair_print },
4096  { NVLIST_DCMD_NAME, NVLIST_DCMD_USAGE, NVLIST_DCMD_DESCR,

```

```

4097         print_nvlist },

4099     /* from pg.c */
4100     { "pg", "?[-q]", "display a pg", pg },

4102     /* from rctl.c */
4103     { "rctl_dict", "?", "print systemwide default rctl definitions",
4104       rctl_dict },
4105     { "rctl_list", ":[handle]", "print rctls for the given proc",
4106       rctl_list },
4107     { "rctl", ":[handle]", "print a rctl_t, only if it matches the handle",
4108       rctl },
4109     { "rctl_validate", ":[-v] [-n #]", "test resource control value "
4110       "sequence", rctl_validate },

4112     /* from subj.c */
4113     { "rwlock", ":", "dump out a readers/writer lock", rwlock },
4114     { "mutex", ":[-f]", "dump out an adaptive or spin mutex", mutex,
4115       mutex_help },
4116     { "sobj2ts", ":", "perform turnstile lookup on synch object", sobj2ts },
4117     { "wchaninfo", "?[-v]", "dump condition variable", wchaninfo },
4118     { "turnstile", "?", "display a turnstile", turnstile },

4120     /* from stream.c */
4121     { "mblk", ":[-q|v] [-f|F flag] [-t|T type] [-l|L|B len] [-d dbaddr]",
4122       "print an mblk", mblk_prt, mblk_help },
4123     { "mblk_verify", "?", "verify integrity of an mblk", mblk_verify },
4124     { "mblk2dblk", ":", "convert mblk_t address to dblk_t address",
4125       mblk2dblk },
4126     { "q2otherq", ":", "print peer queue for a given queue", q2otherq },
4127     { "q2rdq", ":", "print read queue for a given queue", q2rdq },
4128     { "q2syncq", ":", "print syncq for a given queue", q2syncq },
4129     { "q2stream", ":", "print stream pointer for a given queue", q2stream },
4130     { "q2wrq", ":", "print write queue for a given queue", q2wrq },
4131     { "queue", ":[-q|v] [-m mod] [-f flag] [-F flag] [-s syncq_addr]",
4132       "filter and display STREAM queue", queue, queue_help },
4133     { "stdata", ":[-q|v] [-f flag] [-F flag]",
4134       "filter and display STREAM head", stdata, stdata_help },
4135     { "str2mate", ":", "print mate of this stream", str2mate },
4136     { "str2wrq", ":", "print write queue of this stream", str2wrq },
4137     { "stream", ":", "display STREAM", stream },
4138     { "strftevent", ":", "print STREAMS flow trace event", strftevent },
4139     { "syncq", ":[-q|v] [-f flag] [-F flag] [-t type] [-T type]",
4140       "filter and display STREAM sync queue", syncq, syncq_help },
4141     { "syncq2q", ":", "print queue for a given syncq", syncq2q },

4143     /* from taskq.c */
4144     { "taskq", ":[-atT] [-m min_maxq] [-n name]",
4145       "display a taskq", taskq, taskq_help },
4146     { "taskq_entry", ":", "display a taskq_ent_t", taskq_ent },

4148     /* from thread.c */
4149     { "thread", "?[-bdfimps]", "display a summarized kthread_t", thread,
4150       thread_help },
4151     { "threadlist", "?[-t] [-v [count]]",
4152       "display threads and associated C stack traces", threadlist,
4153       threadlist_help },
4154     { "stackinfo", "?[-h|-a]", "display kthread_t stack usage", stackinfo,
4155       stackinfo_help },

4157     /* from tsd.c */
4158     { "tsd", ":-k key", "print tsd[key-l] for this thread", ttotsd },
4159     { "tsdtot", ":", "find thread with this tsd", tsdtot },

4161     /*
4162     * typegraph does not work under kmdb, as it requires too much memory

```

```

4163     * for its internal data structures.
4164     */
4165     #ifndef _KMDB
4166     /* from typegraph.c */
4167     { "findlocks", ":", "find locks held by specified thread", findlocks },
4168     { "findfalse", "?[-v]", "find potentially falsely shared structures",
4169       findfalse },
4170     { "typegraph", NULL, "build type graph", typegraph },
4171     { "istype", ":-type", "manually set object type", istype },
4172     { "notype", ":", "manually clear object type", notype },
4173     { "whattype", ":", "determine object type", whattype },
4174     #endif

4176     /* from vfs.c */
4177     { "fsinfo", "?[-v]", "print mounted filesystems", fsinfo },
4178     { "pfiles", ":[-fp]", "print process file information", pfiles,
4179       pfiles_help },

4181     /* from zone.c */
4182     { "zone", "?[-r [-v]]", "display kernel zone(s)", zoneprt },
4183     { "zsd", ":[-v] [zsd_key]", "display zone-specific-data entries for "
4184       "selected zones", zsd },

4186     { NULL }
4187 };

4189 static const mdb_walker_t walkers[] = {

4191     /* from genunix.c */
4192     { "callouts_bytime", "walk callouts by list chain (expiration time)",
4193       callout_walk_init, callout_walk_step, callout_walk_fini,
4194       (void *)CALLOUT_WALK_BYLIST },
4195     { "callouts_byid", "walk callouts by id hash chain",
4196       callout_walk_init, callout_walk_step, callout_walk_fini,
4197       (void *)CALLOUT_WALK_BYID },
4198     { "callout_list", "walk a callout list", callout_list_walk_init,
4199       callout_list_walk_step, callout_list_walk_fini },
4200     { "callout_table", "walk callout table array", callout_table_walk_init,
4201       callout_table_walk_step, callout_table_walk_fini },
4202     { "cpu", "walk cpu structures", cpu_walk_init, cpu_walk_step },
4203     { "ereportq_dump", "walk list of ereports in dump error queue",
4204       ereportq_dump_walk_init, ereportq_dump_walk_step, NULL },
4205     { "ereportq_pend", "walk list of ereports in pending error queue",
4206       ereportq_pend_walk_init, ereportq_pend_walk_step, NULL },
4207     { "errorq", "walk list of system error queues",
4208       errorq_walk_init, errorq_walk_step, NULL },
4209     { "errorq_data", "walk pending error queue data buffers",
4210       eqd_walk_init, eqd_walk_step, eqd_walk_fini },
4211     { "allfile", "given a proc pointer, list all file pointers",
4212       file_walk_init, allfile_walk_step, file_walk_fini },
4213     { "file", "given a proc pointer, list of open file pointers",
4214       file_walk_init, file_walk_step, file_walk_fini },
4215     { "lock_descriptor", "walk lock_descriptor_t structures",
4216       ld_walk_init, ld_walk_step, NULL },
4217     { "lock_graph", "walk lock graph",
4218       lg_walk_init, lg_walk_step, NULL },
4219     { "port", "given a proc pointer, list of created event ports",
4220       port_walk_init, port_walk_step, NULL },
4221     { "portev", "given a port pointer, list of events in the queue",
4222       portev_walk_init, portev_walk_step, portev_walk_fini },
4223     { "proc", "list of active proc_t structures",
4224       proc_walk_init, proc_walk_step, proc_walk_fini },
4225     { "projects", "walk a list of kernel projects",
4226       project_walk_init, project_walk_step, NULL },
4227     { "sysevent_pend", "walk sysevent pending queue",
4228       sysevent_pend_walk_init, sysevent_walk_step,

```

```

4229         sysevent_walk_fini},
4230     { "sysevent_sent", "walk sysevent sent queue", sysevent_sent_walk_init,
4231       sysevent_walk_step, sysevent_walk_fini},
4232     { "sysevent_channel", "walk sysevent channel subscriptions",
4233       sysevent_channel_walk_init, sysevent_channel_walk_step,
4234       sysevent_channel_walk_fini},
4235     { "sysevent_class_list", "walk sysevent subscription's class list",
4236       sysevent_class_list_walk_init, sysevent_class_list_walk_step,
4237       sysevent_class_list_walk_fini},
4238     { "sysevent_subclass_list",
4239       "walk sysevent subscription's subclass list",
4240       sysevent_subclass_list_walk_init,
4241       sysevent_subclass_list_walk_step,
4242       sysevent_subclass_list_walk_fini},
4243     { "task", "given a task pointer, walk its processes",
4244       task_walk_init, task_walk_step, NULL },

4246 /* from avl.c */
4247 { AVL_WALK_NAME, AVL_WALK_DESC,
4248   avl_walk_init, avl_walk_step, avl_walk_fini },

4250 /* from bio.c */
4251 { "buf", "walk the bio buf hash",
4252   buf_walk_init, buf_walk_step, buf_walk_fini },

4254 /* from contract.c */
4255 { "contract", "walk all contracts, or those of the specified type",
4256   ct_walk_init, generic_walk_step, NULL },
4257 { "ct_event", "walk events on a contract event queue",
4258   ct_event_walk_init, generic_walk_step, NULL },
4259 { "ct_listener", "walk contract event queue listeners",
4260   ct_listener_walk_init, generic_walk_step, NULL },

4262 /* from cpupart.c */
4263 { "cpupart_cpulist", "given an cpupart_t, walk cpus in partition",
4264   cpupart_cpulist_walk_init, cpupart_cpulist_walk_step,
4265   NULL },
4266 { "cpupart_walk", "walk the set of cpu partitions",
4267   cpupart_walk_init, cpupart_walk_step, NULL },

4269 /* from ctxop.c */
4270 { "ctxop", "walk list of context ops on a thread",
4271   ctxop_walk_init, ctxop_walk_step, ctxop_walk_fini },

4273 /* from cyclic.c */
4274 { "cyccpu", "walk per-CPU cyc_cpu structures",
4275   cyccpu_walk_init, cyccpu_walk_step, NULL },
4276 { "cycomni", "for an omnipresent cyclic, walk cyc_omni_cpu list",
4277   cycomni_walk_init, cycomni_walk_step, NULL },
4278 { "cyctrace", "walk cyclic trace buffer",
4279   cyctrace_walk_init, cyctrace_walk_step, cyctrace_walk_fini },

4281 /* from devinfo.c */
4282 { "binding_hash", "walk all entries in binding hash table",
4283   binding_hash_walk_init, binding_hash_walk_step, NULL },
4284 { "devinfo", "walk devinfo tree or subtree",
4285   devinfo_walk_init, devinfo_walk_step, devinfo_walk_fini },
4286 { "devinfo_audit_log", "walk devinfo audit system-wide log",
4287   devinfo_audit_log_walk_init, devinfo_audit_log_walk_step,
4288   devinfo_audit_log_walk_fini},
4289 { "devinfo_audit_node", "walk per-devinfo audit history",
4290   devinfo_audit_node_walk_init, devinfo_audit_node_walk_step,
4291   devinfo_audit_node_walk_fini},
4292 { "devinfo_children", "walk children of devinfo node",
4293   devinfo_children_walk_init, devinfo_children_walk_step,
4294   devinfo_children_walk_fini },

```

```

4295     { "devinfo_parents", "walk ancestors of devinfo node",
4296       devinfo_parents_walk_init, devinfo_parents_walk_step,
4297       devinfo_parents_walk_fini },
4298     { "devinfo_siblings", "walk siblings of devinfo node",
4299       devinfo_siblings_walk_init, devinfo_siblings_walk_step, NULL },
4300     { "devi_next", "walk devinfo list",
4301       NULL, devi_next_walk_step, NULL },
4302     { "devnames", "walk devnames array",
4303       devnames_walk_init, devnames_walk_step, devnames_walk_fini },
4304     { "minornode", "given a devinfo node, walk minor nodes",
4305       minornode_walk_init, minornode_walk_step, NULL },
4306     { "softstate",
4307       "given an i_ddi_soft_state*, list all in-use driver stateps",
4308       soft_state_walk_init, soft_state_walk_step,
4309       NULL, NULL },
4310     { "softstate_all",
4311       "given an i_ddi_soft_state*, list all driver stateps",
4312       soft_state_walk_init, soft_state_all_walk_step,
4313       NULL, NULL },
4314     { "devinfo_fmc",
4315       "walk a fault management handle cache active list",
4316       devinfo_fmc_walk_init, devinfo_fmc_walk_step, NULL },

4318 /* from group.c */
4319 { "group", "walk all elements of a group",
4320   group_walk_init, group_walk_step, NULL },

4322 /* from irm.c */
4323 { "irmpools", "walk global list of interrupt pools",
4324   irmpools_walk_init, list_walk_step, list_walk_fini },
4325 { "irmreqs", "walk list of interrupt requests in an interrupt pool",
4326   irmreqs_walk_init, list_walk_step, list_walk_fini },

4328 /* from kmem.c */
4329 { "allocdby", "given a thread, walk its allocated bufctls",
4330   allocdby_walk_init, allocdby_walk_step, allocdby_walk_fini },
4331 { "bufctl", "walk a kmem cache's bufctls",
4332   bufctl_walk_init, kmem_walk_step, kmem_walk_fini },
4333 { "bufctl_history", "walk the available history of a bufctl",
4334   bufctl_history_walk_init, bufctl_history_walk_step,
4335   bufctl_history_walk_fini },
4336 { "freedby", "given a thread, walk its freed bufctls",
4337   freedby_walk_init, allocdby_walk_step, allocdby_walk_fini },
4338 { "freectl", "walk a kmem cache's free bufctls",
4339   freectl_walk_init, kmem_walk_step, kmem_walk_fini },
4340 { "freectl_constructed", "walk a kmem cache's constructed free bufctls",
4341   freectl_constructed_walk_init, kmem_walk_step, kmem_walk_fini },
4342 { "freemem", "walk a kmem cache's free memory",
4343   freemem_walk_init, kmem_walk_step, kmem_walk_fini },
4344 { "freemem_constructed", "walk a kmem cache's constructed free memory",
4345   freemem_constructed_walk_init, kmem_walk_step, kmem_walk_fini },
4346 { "kmem", "walk a kmem cache",
4347   kmem_walk_init, kmem_walk_step, kmem_walk_fini },
4348 { "kmem_cpu_cache", "given a kmem cache, walk its per-CPU caches",
4349   kmem_cpu_cache_walk_init, kmem_cpu_cache_walk_step, NULL },
4350 { "kmem_hash", "given a kmem cache, walk its allocated hash table",
4351   kmem_hash_walk_init, kmem_hash_walk_step, kmem_hash_walk_fini },
4352 { "kmem_log", "walk the kmem transaction log",
4353   kmem_log_walk_init, kmem_log_walk_step, kmem_log_walk_fini },
4354 { "kmem_slab", "given a kmem cache, walk its slabs",
4355   kmem_slab_walk_init, combined_walk_step, combined_walk_fini },
4356 { "kmem_slab_partial",
4357   "given a kmem cache, walk its partially allocated slabs (min 1)",
4358   kmem_slab_walk_partial_init, combined_walk_step,
4359   combined_walk_fini },
4360 { "vmem", "walk vmem structures in pre-fix, depth-first order",

```



```

4361         vmem_walk_init, vmem_walk_step, vmem_walk_fini },
4362     { "vmem_alloc", "given a vmem_t, walk its allocated vmem_segs",
4363       vmem_alloc_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4364     { "vmem_free", "given a vmem_t, walk its free vmem_segs",
4365       vmem_free_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4366     { "vmem_postfix", "walk vmem structures in post-fix, depth-first order",
4367       vmem_walk_init, vmem_postfix_walk_step, vmem_walk_fini },
4368     { "vmem_seg", "given a vmem_t, walk all of its vmem_segs",
4369       vmem_seg_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },
4370     { "vmem_span", "given a vmem_t, walk its spanning vmem_segs",
4371       vmem_span_walk_init, vmem_seg_walk_step, vmem_seg_walk_fini },

4373     /* from ldi.c */
4374     { "ldi_handle", "walk the layered driver handle hash",
4375       ldi_handle_walk_init, ldi_handle_walk_step, NULL },
4376     { "ldi_ident", "walk the layered driver identifier hash",
4377       ldi_ident_walk_init, ldi_ident_walk_step, NULL },

4379     /* from leaky.c + leaky_subr.c */
4380     { "leak", "given a leaked bufctl or vmem_seg, find leaks w/ same "
4381       "stack trace",
4382       leaky_walk_init, leaky_walk_step, leaky_walk_fini },
4383     { "leakbuf", "given a leaked bufctl or vmem_seg, walk buffers for "
4384       "leaks w/ same stack trace",
4385       leaky_walk_init, leaky_buf_walk_step, leaky_walk_fini },

4387     /* from lgrp.c */
4388     { "lgrp_cpulist", "walk CPUs in a given lgroup",
4389       lgrp_cpulist_walk_init, lgrp_cpulist_walk_step, NULL },
4390     { "lgrptbl", "walk lgroup table",
4391       lgrp_walk_init, lgrp_walk_step, NULL },
4392     { "lgrp_parents", "walk up lgroup lineage from given lgroup",
4393       lgrp_parents_walk_init, lgrp_parents_walk_step, NULL },
4394     { "lgrp_rsrc_mem", "walk lgroup memory resources of given lgroup",
4395       lgrp_rsrc_mem_walk_init, lgrp_set_walk_step, NULL },
4396     { "lgrp_rsrc_cpu", "walk lgroup CPU resources of given lgroup",
4397       lgrp_rsrc_cpu_walk_init, lgrp_set_walk_step, NULL },

4399     /* from list.c */
4400     { LIST_WALK_NAME, LIST_WALK_DESC,
4401       list_walk_init, list_walk_step, list_walk_fini },

4403     /* from mdi.c */
4404     { "mdipi_client_list", "Walker for mdi_pathinfo pi_client_link",
4405       mdi_pi_client_link_walk_init,
4406       mdi_pi_client_link_walk_step,
4407       mdi_pi_client_link_walk_fini },
4408     { "mdipi_phci_list", "Walker for mdi_pathinfo pi_phci_link",
4409       mdi_pi_phci_link_walk_init,
4410       mdi_pi_phci_link_walk_step,
4411       mdi_pi_phci_link_walk_fini },
4412     { "mdiphci_list", "Walker for mdi_phci ph_next link",
4413       mdi_phci_ph_next_walk_init,
4414       mdi_phci_ph_next_walk_step,
4415       mdi_phci_ph_next_walk_fini },

4417     /* from memory.c */
4418     { "allpages", "walk all pages, including free pages",
4419       allpages_walk_init, allpages_walk_step, allpages_walk_fini },
4420     { "anon", "given an amp, list allocated anon structures",
4421       anon_walk_init, anon_walk_step, anon_walk_fini,
4422       ANON_WALK_ALLOC },
4423     { "anon_all", "given an amp, list contents of all anon slots",
4424       anon_walk_init, anon_walk_step, anon_walk_fini,
4425       ANON_WALK_ALL },
4426     { "memlist", "walk specified memlist",

```

```

4427         NULL, memlist_walk_step, NULL },
4428     { "page", "walk all pages, or those from the specified vnode",
4429       page_walk_init, page_walk_step, page_walk_fini },
4430     { "seg", "given an as, list of segments",
4431       seg_walk_init, avl_walk_step, avl_walk_fini },
4432     { "segvn_anon",
4433       "given a struct segvn_data, list allocated anon structures",
4434       segvn_anon_walk_init, anon_walk_step, anon_walk_fini,
4435       ANON_WALK_ALLOC },
4436     { "segvn_anon_all",
4437       "given a struct segvn_data, list contents of all anon slots",
4438       segvn_anon_walk_init, anon_walk_step, anon_walk_fini,
4439       ANON_WALK_ALL },
4440     { "segvn_pages",
4441       "given a struct segvn_data, list resident pages in "
4442       "offset order",
4443       segvn_pages_walk_init, segvn_pages_walk_step,
4444       segvn_pages_walk_fini, SEGVN_PAGES_RESIDENT },
4445     { "segvn_pages_all",
4446       "for each offset in a struct segvn_data, give page_t pointer "
4447       "(if resident), or NULL.",
4448       segvn_pages_walk_init, segvn_pages_walk_step,
4449       segvn_pages_walk_fini, SEGVN_PAGES_ALL },
4450     { "swapinfo", "walk swapinfo structures",
4451       swap_walk_init, swap_walk_step, NULL },

4453     /* from mmd.c */
4454     { "pattr", "walk pattr_t structures", pattr_walk_init,
4455       mmdq_walk_step, mmdq_walk_fini },
4456     { "pdesc", "walk pdesc_t structures",
4457       pdesc_walk_init, mmdq_walk_step, mmdq_walk_fini },
4458     { "pdesc_slab", "walk pdesc_slab_t structures",
4459       pdesc_slab_walk_init, mmdq_walk_step, mmdq_walk_fini },

4461     /* from modhash.c */
4462     { "modhash", "walk list of mod_hash structures", modhash_walk_init,
4463       modhash_walk_step, NULL },
4464     { "modent", "walk list of entries in a given mod_hash",
4465       modent_walk_init, modent_walk_step, modent_walk_fini },
4466     { "modchain", "walk list of entries in a given mod_hash_entry",
4467       NULL, modchain_walk_step, NULL },

4469     /* from net.c */
4470     { "icmp", "walk ICMP control structures using MI for all stacks",
4471       mi_payload_walk_init, mi_payload_walk_step, NULL,
4472       &mi_icmp_arg },
4473     { "mi", "given a MI_O, walk the MI",
4474       mi_walk_init, mi_walk_step, mi_walk_fini, NULL },
4475     { "sonode", "given a sonode, walk its children",
4476       sonode_walk_init, sonode_walk_step, sonode_walk_fini, NULL },
4477     { "icmp_stacks", "walk all the icmp_stack_t",
4478       icmp_stacks_walk_init, icmp_stacks_walk_step, NULL },
4479     { "tcp_stacks", "walk all the tcp_stack_t",
4480       tcp_stacks_walk_init, tcp_stacks_walk_step, NULL },
4481     { "udp_stacks", "walk all the udp_stack_t",
4482       udp_stacks_walk_init, udp_stacks_walk_step, NULL },
4483     { "dccp_stacks", "walk all the dccp_stack_t",
4484       dccp_stacks_walk_init, dccp_stacks_walk_step, NULL },
4485 #endif /* ! codereview */

4487     /* from netstack.c */
4488     { "netstack", "walk a list of kernel netstacks",
4489       netstack_walk_init, netstack_walk_step, NULL },

4491     /* from nvpair.c */
4492     { NVPAIR_WALKER_NAME, NVPAIR_WALKER_DESCR,

```

```

4493         nvpair_walk_init, nvpair_walk_step, NULL },

4495     /* from rctl.c */
4496     { "rctl_dict_list", "walk all rctl_dict_entry_t's from rctl_lists",
4497       rctl_dict_walk_init, rctl_dict_walk_step, NULL },
4498     { "rctl_set", "given a rctl_set, walk all rctls", rctl_set_walk_init,
4499       rctl_set_walk_step, NULL },
4500     { "rctl_val", "given a rctl_t, walk all rctl_val entries associated",
4501       rctl_val_walk_init, rctl_val_walk_step },

4503     /* from sobj.c */
4504     { "blocked", "walk threads blocked on a given sobj",
4505       blocked_walk_init, blocked_walk_step, NULL },
4506     { "wchan", "given a wchan, list of blocked threads",
4507       wchan_walk_init, wchan_walk_step, wchan_walk_fini },

4509     /* from stream.c */
4510     { "b_cont", "walk mblk_t list using b_cont",
4511       mblk_walk_init, b_cont_step, mblk_walk_fini },
4512     { "b_next", "walk mblk_t list using b_next",
4513       mblk_walk_init, b_next_step, mblk_walk_fini },
4514     { "qlink", "walk queue_t list using q_link",
4515       queue_walk_init, queue_link_step, queue_walk_fini },
4516     { "qnext", "walk queue_t list using q_next",
4517       queue_walk_init, queue_next_step, queue_walk_fini },
4518     { "strftblk", "given a dblk_t, walk STREAMS flow trace event list",
4519       strftblk_walk_init, strftblk_step, strftblk_walk_fini },
4520     { "readq", "walk read queue side of stdata",
4521       str_walk_init, str_read_step, str_walk_fini },
4522     { "writeq", "walk write queue side of stdata",
4523       str_walk_init, str_write_step, str_walk_fini },

4525     /* from taskq.c */
4526     { "taskq_thread", "given a taskq_t, list all of its threads",
4527       taskq_thread_walk_init,
4528       taskq_thread_walk_step,
4529       taskq_thread_walk_fini },
4530     { "taskq_entry", "given a taskq_t*, list all taskq_ent_t in the list",
4531       taskq_ent_walk_init, taskq_ent_walk_step, NULL },

4533     /* from thread.c */
4534     { "deathrow", "walk threads on both lwp_ and thread_deathrow",
4535       deathrow_walk_init, deathrow_walk_step, NULL },
4536     { "cpu_dispg", "given a cpu_t, walk threads in dispatcher queues",
4537       cpu_dispg_walk_init, dispq_walk_step, dispq_walk_fini },
4538     { "cpupart_dispg",
4539       "given a cpupart_t, walk threads in dispatcher queues",
4540       cpupart_dispg_walk_init, dispq_walk_step, dispq_walk_fini },
4541     { "lwp_deathrow", "walk lwp_deathrow",
4542       lwp_deathrow_walk_init, deathrow_walk_step, NULL },
4543     { "thread", "global or per-process kthread_t structures",
4544       thread_walk_init, thread_walk_step, thread_walk_fini },
4545     { "thread_deathrow", "walk threads on thread_deathrow",
4546       thread_deathrow_walk_init, deathrow_walk_step, NULL },

4548     /* from tsd.c */
4549     { "tsd", "walk list of thread-specific data",
4550       tsd_walk_init, tsd_walk_step, tsd_walk_fini },

4552     /* from tsol.c */
4553     { "tnrh", "walk remote host cache structures",
4554       tn timer_walk_init, tn timer_walk_step, tn timer_walk_fini },
4555     { "tnrhtp", "walk remote host template structures",
4556       tn timer_walk_init, tn timer_walk_step, tn timer_walk_fini },

4558     /*

```

```

4559     * typegraph does not work under kmdb, as it requires too much memory
4560     * for its internal data structures.
4561     */
4562 #ifndef _KMDB
4563     /* from typegraph.c */
4564     { "typeconflict", "walk buffers with conflicting type inferences",
4565       typegraph_walk_init, typeconflict_walk_step },
4566     { "typeunknown", "walk buffers with unknown types",
4567       typegraph_walk_init, typeunknown_walk_step },
4568 #endif

4570     /* from vfs.c */
4571     { "vfs", "walk file system list",
4572       vfs_walk_init, vfs_walk_step },

4574     /* from zone.c */
4575     { "zone", "walk a list of kernel zones",
4576       zone_walk_init, zone_walk_step, NULL },
4577     { "zsd", "walk list of zsd entries for a zone",
4578       zsd_walk_init, zsd_walk_step, NULL },

4580     { NULL }
4581 };

4583 static const mdb_modinfo_t modinfo = { MDB_API_VERSION, dcmsgs, walkers };

4585 /*ARGSUSED*/
4586 static void
4587 genunix_statechange_cb(void *ignored)
4588 {
4589     /*
4590      * Force ::findleaks and ::stacks to let go any cached state.
4591      */
4592     leaky_cleanup(1);
4593     stacks_cleanup(1);

4595     kmem_statechange(); /* notify kmem */
4596 }

4598 const mdb_modinfo_t *
4599 _mdb_init(void)
4600 {
4601     kmem_init();

4603     (void) mdb_callback_add(MDB_CALLBACK_STCHG,
4604       genunix_statechange_cb, NULL);

4606     return (&modinfo);
4607 }

4609 void
4610 _mdb_fini(void)
4611 {
4612     leaky_cleanup(1);
4613     stacks_cleanup(1);
4614 }

```

new/usr/src/cmd/mdb/common/modules/genunix/net.c

1

```
*****
43063 Sat Aug 18 10:37:01 2012
new/usr/src/cmd/mdb/common/modules/genunix/net.c
dccc: build fixes, mdb (vfs sonode missing)
*****

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 <mdb/mdb_modapi.h>
27 #include <mdb/mdb_ks.h>
28 #include <mdb/mdb_ctf.h>
29 #include <sys/types.h>
30 #include <sys/tihdr.h>
31 #include <inet/led.h>
32 #include <inet/common.h>
33 #include <netinet/in.h>
34 #include <netinet/ip6.h>
35 #include <netinet/icmp6.h>
36 #include <inet/ip.h>
37 #include <inet/ip6.h>
38 #include <inet/ipclassifier.h>
39 #include <inet/tcp.h>
40 #include <sys/stream.h>
41 #include <sys/vfs.h>
42 #include <sys/stropts.h>
43 #include <sys/tpicommon.h>
44 #include <sys/socket.h>
45 #include <sys/socketvar.h>
46 #include <sys/cred_impl.h>
47 #include <inet/udp_impl.h>
48 #include <inet/rawip_impl.h>
49 #include <inet/mi.h>
50 #include <inet/dccp_impl.h>
51 #endif /* ! codereview */
52 #include <fs/sockfs/socktpi_impl.h>
53 #include <net/bridge_impl.h>
54 #include <io/trill_impl.h>
55 #include <sys/mac_impl.h>

57 #define ADDR_V6_WIDTH 23
58 #define ADDR_V4_WIDTH 15

60 #define NETSTAT_ALL 0x01
61 #define NETSTAT_VERBOSE 0x02
```

new/usr/src/cmd/mdb/common/modules/genunix/net.c

2

```
62 #define NETSTAT_ROUTE 0x04
63 #define NETSTAT_V4 0x08
64 #define NETSTAT_V6 0x10
65 #define NETSTAT_UNIX 0x20

67 #define NETSTAT_FIRST 0x80000000u

69 typedef struct netstat_cb_data_s {
70     uint_t opts;
71     conn_t conn;
72     int af;
73 } netstat_cb_data_t;

75 int
76 icmp_stacks_walk_init(mdb_walk_state_t *wsp)
77 {
78     if (mdb_layered_walk("netstack", wsp) == -1) {
79         mdb_warn("can't walk 'netstack'");
80         return (WALK_ERR);
81     }
82     return (WALK_NEXT);
83 }

85 int
86 icmp_stacks_walk_step(mdb_walk_state_t *wsp)
87 {
88     uintptr_t kaddr;
89     netstack_t nss;

91     if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
92         mdb_warn("can't read netstack at %p", wsp->walk_addr);
93         return (WALK_ERR);
94     }
95     kaddr = (uintptr_t)nss.netstack_modules[NS_ICMP];
96     return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
97 }

99 int
100 tcp_stacks_walk_init(mdb_walk_state_t *wsp)
101 {
102     if (mdb_layered_walk("netstack", wsp) == -1) {
103         mdb_warn("can't walk 'netstack'");
104         return (WALK_ERR);
105     }
106     return (WALK_NEXT);
107 }

109 int
110 tcp_stacks_walk_step(mdb_walk_state_t *wsp)
111 {
112     uintptr_t kaddr;
113     netstack_t nss;

115     if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
116         mdb_warn("can't read netstack at %p", wsp->walk_addr);
117         return (WALK_ERR);
118     }
119     kaddr = (uintptr_t)nss.netstack_modules[NS_TCP];
120     return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
121 }

123 int
124 udp_stacks_walk_init(mdb_walk_state_t *wsp)
125 {
126     if (mdb_layered_walk("netstack", wsp) == -1) {
127         mdb_warn("can't walk 'netstack'");
```

```

128         return (WALK_ERR);
129     }
130     return (WALK_NEXT);
131 }

133 int
134 udp_stacks_walk_step(mdb_walk_state_t *wsp)
135 {
136     uintptr_t kaddr;
137     netstack_t nss;

139     if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
140         mdb_warn("can't read netstack at %p", wsp->walk_addr);
141         return (WALK_ERR);
142     }
143     kaddr = (uintptr_t)nss.netstack_modules[NS_UDP];
144     return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
145 }

147 int
148 dccp_stacks_walk_init(mdb_walk_state_t *wsp)
149 {
150     if (mdb_layered_walk("netstack", wsp) == -1) {
151         mdb_warn("can't walk 'netstack'");
152         return (WALK_ERR);
153     }
154     return (WALK_NEXT);
155 }

157 int
158 dccp_stacks_walk_step(mdb_walk_state_t *wsp)
159 {
160     uintptr_t kaddr;
161     netstack_t nss;

163     if (mdb_vread(&nss, sizeof (nss), wsp->walk_addr) == -1) {
164         mdb_warn("can't read netstack at %p", wsp->walk_addr);
165         return (WALK_ERR);
166     }
167     kaddr = (uintptr_t)nss.netstack_modules[NS_DCCP];
168     return (wsp->walk_callback(kaddr, wsp->walk_layer, wsp->walk_cbdata));
169 }

171 #endif /* ! codereview */
172 /*
173  * Print an IPv4 address and port number in a compact and easy to read format
174  * The arguments are in network byte order
175  */
176 static void
177 net_ipv4addrport_pr(const in6_addr_t *nipv6addr, in_port_t nport)
178 {
179     uint32_t naddr = V4_PART_OF_V6((*nipv6addr));

181     mdb_nhconvert(&nport, &nport, sizeof (nport));
182     mdb_printf("%*I.%-5hu", ADDR_V4_WIDTH, naddr, nport);
183 }

185 /*
186  * Print an IPv6 address and port number in a compact and easy to read format
187  * The arguments are in network byte order
188  */
189 static void
190 net_ipv6addrport_pr(const in6_addr_t *naddr, in_port_t nport)
191 {
192     mdb_nhconvert(&nport, &nport, sizeof (nport));
193     mdb_printf("%*N.%-5hu", ADDR_V6_WIDTH, naddr, nport);

```

```

194 }

196 static int
197 net_tcp_active(const tcp_t *tcp)
198 {
199     return (tcp->tcp_state >= TCPS_ESTABLISHED);
200 }

202 static int
203 net_tcp_ipv4(const tcp_t *tcp)
204 {
205     return ((tcp->tcp_connp->conn_ipversion == IPV4_VERSION) ||
206         (IN6_IS_ADDR_UNSPECIFIED(&tcp->tcp_connp->conn_laddr_v6) &&
207         (tcp->tcp_state <= TCPS_LISTEN)));
208 }

210 static int
211 net_tcp_ipv6(const tcp_t *tcp)
212 {
213     return (tcp->tcp_connp->conn_ipversion == IPV6_VERSION);
214 }

216 static int
217 net_udp_active(const udp_t *udp)
218 {
219     return ((udp->udp_state == TS_IDLE) ||
220         (udp->udp_state == TS_DATA_XFER));
221 }

223 static int
224 net_udp_ipv4(const udp_t *udp)
225 {
226     return ((udp->udp_connp->conn_ipversion == IPV4_VERSION) ||
227         (IN6_IS_ADDR_UNSPECIFIED(&udp->udp_connp->conn_laddr_v6) &&
228         (udp->udp_state <= TS_IDLE)));
229 }

231 static int
232 net_udp_ipv6(const udp_t *udp)
233 {
234     return (udp->udp_connp->conn_ipversion == IPV6_VERSION);
235 }

237 static int
238 net_dccp_active(const dccp_t *dccp)
239 {
240     return ((dccp->dccp_state == TS_IDLE) ||
241         (dccp->dccp_state == TS_DATA_XFER));
242 }

244 static int
245 net_dccp_ipv4(const dccp_t *dccp)
246 {
247     return ((dccp->dccp_connp->conn_ipversion == IPV4_VERSION) ||
248         (IN6_IS_ADDR_UNSPECIFIED(&dccp->dccp_connp->conn_laddr_v6) &&
249         (dccp->dccp_state <= DCCPS_LISTEN)));
250 }

252 static int
253 net_dccp_ipv6(const dccp_t *dccp)
254 {
255     return (dccp->dccp_connp->conn_ipversion == IPV6_VERSION);
256 }

258 #endif /* ! codereview */
259 int

```

```

260 sonode_walk_init(mdb_walk_state_t *wsp)
261 {
262     if (wsp->walk_addr == NULL) {
263         GElf_Sym sym;
264         struct socklist *slp;
265
266         if (mdb_lookup_by_obj("sockfs", "socklist", &sym) == -1) {
267             mdb_warn("failed to lookup sockfs'socklist");
268             return (WALK_ERR);
269         }
270
271         slp = (struct socklist *) (uintptr_t) sym.st_value;
272
273         if (mdb_vread(&wsp->walk_addr, sizeof (wsp->walk_addr),
274             (uintptr_t) &slp->sl_list) == -1) {
275             mdb_warn("failed to read address of initial sonode "
276                 "at %p", &slp->sl_list);
277             return (WALK_ERR);
278         }
279     }
280
281     wsp->walk_data = mdb_alloc(sizeof (struct sotpi_sonode), UM_SLEEP);
282     return (WALK_NEXT);
283 }
284
285 int
286 sonode_walk_step(mdb_walk_state_t *wsp)
287 {
288     int status;
289     struct sotpi_sonode *stp;
290
291     if (wsp->walk_addr == NULL)
292         return (WALK_DONE);
293
294     if (mdb_vread(wsp->walk_data, sizeof (struct sotpi_sonode),
295         wsp->walk_addr) == -1) {
296         mdb_warn("failed to read sonode at %p", wsp->walk_addr);
297         return (WALK_ERR);
298     }
299
300     status = wsp->walk_callback(wsp->walk_addr, wsp->walk_data,
301         wsp->walk_cbdata);
302
303     stp = wsp->walk_data;
304
305     wsp->walk_addr = (uintptr_t) stp->st_info.sti_next_so;
306     return (status);
307 }
308
309 void
310 sonode_walk_fini(mdb_walk_state_t *wsp)
311 {
312     mdb_free(wsp->walk_data, sizeof (struct sotpi_sonode));
313 }
314
315 struct mi_walk_data {
316     uintptr_t mi_wd_miofirst;
317     MI_O mi_wd_miodata;
318 };
319
320 int
321 mi_walk_init(mdb_walk_state_t *wsp)
322 {
323     struct mi_walk_data *wdp;
324
325     if (wsp->walk_addr == NULL) {

```

```

326         mdb_warn("mi doesn't support global walks\n");
327         return (WALK_ERR);
328     }
329
330     wdp = mdb_alloc(sizeof (struct mi_walk_data), UM_SLEEP);
331
332     /* So that we do not immediately return WALK_DONE below */
333     wdp->mi_wd_miofirst = NULL;
334
335     wsp->walk_data = wdp;
336     return (WALK_NEXT);
337 }
338
339 int
340 mi_walk_step(mdb_walk_state_t *wsp)
341 {
342     struct mi_walk_data *wdp = wsp->walk_data;
343     MI_OP miop = &wdp->mi_wd_miodata;
344     int status;
345
346     /* Always false in the first iteration */
347     if ((wsp->walk_addr == (uintptr_t) NULL) ||
348         (wsp->walk_addr == wdp->mi_wd_miofirst)) {
349         return (WALK_DONE);
350     }
351
352     if (mdb_vread(miop, sizeof (MI_O), wsp->walk_addr) == -1) {
353         mdb_warn("failed to read MI object at %p", wsp->walk_addr);
354         return (WALK_ERR);
355     }
356
357     /* Only true in the first iteration */
358     if (wdp->mi_wd_miofirst == NULL) {
359         wdp->mi_wd_miofirst = wsp->walk_addr;
360         status = WALK_NEXT;
361     } else {
362         status = wsp->walk_callback(wsp->walk_addr + sizeof (MI_O),
363             &miop[1], wsp->walk_cbdata);
364     }
365
366     wsp->walk_addr = (uintptr_t) miop->mi_o_next;
367     return (status);
368 }
369
370 void
371 mi_walk_fini(mdb_walk_state_t *wsp)
372 {
373     mdb_free(wsp->walk_data, sizeof (struct mi_walk_data));
374 }
375
376 typedef struct mi_payload_walk_arg_s {
377     const char *mi_pwa_walker; /* Underlying walker */
378     const off_t mi_pwa_head_off; /* Offset for mi_o_head_t * in stack */
379     const size_t mi_pwa_size; /* size of mi payload */
380     const uint_t mi_pwa_flags; /* device and/or module */
381 } mi_payload_walk_arg_t;
382
383 #define MI_PAYLOAD_DEVICE 0x1
384 #define MI_PAYLOAD_MODULE 0x2
385
386 int
387 mi_payload_walk_init(mdb_walk_state_t *wsp)
388 {
389     const mi_payload_walk_arg_t *arg = wsp->walk_arg;
390
391     if (mdb_layered_walk(arg->mi_pwa_walker, wsp) == -1) {

```

```

392         mdb_warn("can't walk '%s'", arg->mi_pwa_walker);
393         return (WALK_ERR);
394     }
395     return (WALK_NEXT);
396 }

398 int
399 mi_payload_walk_step(mdb_walk_state_t *wsp)
400 {
401     const mi_payload_walk_arg_t *arg = wsp->walk_arg;
402     uintptr_t kaddr;

404     kaddr = wsp->walk_addr + arg->mi_pwa_head_off;

406     if (mdb_vread(&kaddr, sizeof (kaddr), kaddr) == -1) {
407         mdb_warn("can't read address of mi head at %p for %s",
408             kaddr, arg->mi_pwa_walker);
409         return (WALK_ERR);
410     }

412     if (kaddr == 0) {
413         /* Empty list */
414         return (WALK_DONE);
415     }

417     if (mdb_pwalk("genunix'mi", wsp->walk_callback,
418         wsp->walk_cbdata, kaddr) == -1) {
419         mdb_warn("failed to walk genunix'mi");
420         return (WALK_ERR);
421     }
422     return (WALK_NEXT);
423 }

425 const mi_payload_walk_arg_t mi_icmp_arg = {
426     "icmp_stacks", OFFSETOF(icmp_stack_t, is_head), sizeof (icmp_t),
427     MI_PAYLOAD_DEVICE | MI_PAYLOAD_MODULE
428 };

430 int
431 sonode(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
432 {
433     const char *optf = NULL;
434     const char *optt = NULL;
435     const char *optp = NULL;
436     int family, type, proto;
437     int filter = 0;
438     struct sonode so;

440     if (!(flags & DCMD_ADDRSPEC)) {
441         if (mdb_walk_dcmd("genunix'sonode", "genunix'sonode", argc,
442             argv) == -1) {
443             mdb_warn("failed to walk sonode");
444             return (DCMD_ERR);
445         }
447         return (DCMD_OK);
448     }

450     if (mdb_getopts(argc, argv,
451         'f', MDB_OPT_STR, &optf,
452         't', MDB_OPT_STR, &optt,
453         'p', MDB_OPT_STR, &optp,
454         NULL) != argc)
455         return (DCMD_USAGE);

457     if (optf != NULL) {

```

```

458         if (strcmp("inet", optf) == 0)
459             family = AF_INET;
460         else if (strcmp("inet6", optf) == 0)
461             family = AF_INET6;
462         else if (strcmp("unix", optf) == 0)
463             family = AF_UNIX;
464         else
465             family = mdb_strtoull(optf);
466         filter = 1;
467     }

469     if (optt != NULL) {
470         if (strcmp("stream", optt) == 0)
471             type = SOCK_STREAM;
472         else if (strcmp("dgram", optt) == 0)
473             type = SOCK_DGRAM;
474         else if (strcmp("raw", optt) == 0)
475             type = SOCK_RAW;
476         else
477             type = mdb_strtoull(optt);
478         filter = 1;
479     }

481     if (optp != NULL) {
482         proto = mdb_strtoull(optp);
483         filter = 1;
484     }

486     if (DCMD_HDRSPEC(flags) && !filter) {
487         mdb_printf("<u>%-?s Family Type Proto State Mode Flag "
488             "AccessVP%<u>\n", "Sonode:");
489     }

491     if (mdb_vread(&so, sizeof (so), addr) == -1) {
492         mdb_warn("failed to read sonode at %p", addr);
493         return (DCMD_ERR);
494     }

496     if ((optf != NULL) && (so.so_family != family))
497         return (DCMD_OK);

499     if ((optt != NULL) && (so.so_type != type))
500         return (DCMD_OK);

502     if ((optp != NULL) && (so.so_protocol != proto))
503         return (DCMD_OK);

505     if (filter) {
506         mdb_printf("%0?p\n", addr);
507         return (DCMD_OK);
508     }

510     mdb_printf("%0?p ", addr);

512     switch (so.so_family) {
513     case AF_UNIX:
514         mdb_printf("unix ");
515         break;
516     case AF_INET:
517         mdb_printf("inet ");
518         break;
519     case AF_INET6:
520         mdb_printf("inet6 ");
521         break;
522     default:
523         mdb_printf("%6hi", so.so_family);

```

```

524     }

526     switch (so.so_type) {
527     case SOCK_STREAM:
528         mdb_printf(" strm");
529         break;
530     case SOCK_DGRAM:
531         mdb_printf(" dgrm");
532         break;
533     case SOCK_RAW:
534         mdb_printf(" raw ");
535         break;
536     default:
537         mdb_printf(" %4hi", so.so_type);
538     }

540     mdb_printf(" %5hi %05x %04x %04hx\n",
541         so.so_protocol, so.so_state, so.so_mode,
542         so.so_flag);

544     return (DCMD_OK);
545 }

547 #define MI_PAYLOAD      0x1
548 #define MI_DEVICE       0x2
549 #define MI_MODULE       0x4

551 int
552 mi(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
553 {
554     uint_t opts = 0;
555     MI_O     mio;

557     if (!(flags & DCMD_ADDRSPEC))
558         return (DCMD_USAGE);

560     if (mdb_getopts(argc, argv,
561         'p', MDB_OPT_SETBITS, MI_PAYLOAD, &opts,
562         'd', MDB_OPT_SETBITS, MI_DEVICE, &opts,
563         'm', MDB_OPT_SETBITS, MI_MODULE, &opts,
564         NULL) != argc)
565         return (DCMD_USAGE);

567     if ((opts & (MI_DEVICE | MI_MODULE)) == (MI_DEVICE | MI_MODULE)) {
568         mdb_warn("at most one filter, d for devices or m "
569             "for modules, may be specified\n");
570         return (DCMD_USAGE);
571     }

573     if ((opts == 0) && (DCMD_HDRSPEC(flags))) {
574         mdb_printf("%<u>%-?s %-?s %-?s IsDev Dev%<u>\n",
575             "MI_O", "Next", "Prev");
576     }

578     if (mdb_vread(&mio, sizeof (mio), addr) == -1) {
579         mdb_warn("failed to read mi object MI_O at %p", addr);
580         return (DCMD_ERR);
581     }

583     if (opts != 0) {
584         if (mio.mi_o_isdev == B_FALSE) {
585             /* mio is a module */
586             if (!(opts & MI_MODULE) && (opts & MI_DEVICE))
587                 return (DCMD_OK);
588         } else {
589             /* mio is a device */

```

```

590         if (!(opts & MI_DEVICE) && (opts & MI_MODULE))
591             return (DCMD_OK);
592     }

594     if (opts & MI_PAYLOAD)
595         mdb_printf("%p\n", addr + sizeof (MI_O));
596     else
597         mdb_printf("%p\n", addr);
598     return (DCMD_OK);
599 }

601     mdb_printf("%0?p %0?p %0?p ", addr, mio.mi_o_next, mio.mi_o_prev);

603     if (mio.mi_o_isdev == B_FALSE)
604         mdb_printf("FALSE");
605     else
606         mdb_printf("TRUE ");

608     mdb_printf(" %0?p\n", mio.mi_o_dev);

610     return (DCMD_OK);
611 }

613 static int
614 ns_to_stackid(uintptr_t kaddr)
615 {
616     netstack_t nss;

618     if (mdb_vread(&nss, sizeof (nss), kaddr) == -1) {
619         mdb_warn("failed to read netstack_t %p", kaddr);
620         return (0);
621     }
622     return (nss.netstack_stackid);
623 }

627 static void
628 netstat_tcp_verbose_pr(const tcp_t *tcp)
629 {
630     mdb_printf("          %5i %08x %08x %5i %08x %08x %5li %5i\n",
631         tcp->tcp_swnd, tcp->tcp_snxt, tcp->tcp_suna, tcp->tcp_rwnd,
632         tcp->tcp_rack, tcp->tcp_rnxt, tcp->tcp_rto, tcp->tcp_mss);
633 }

635 /*ARGSUSED*/
636 static int
637 netstat_tcp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
638 {
639     netstat_cb_data_t *ncb = cb_data;
640     uint_t opts = ncb->opts;
641     int af = ncb->af;
642     uintptr_t tcp_kaddr;
643     conn_t *connp = &ncb->conn;
644     tcp_t tcps, *tcp;

646     if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
647         mdb_warn("failed to read conn_t at %p", kaddr);
648         return (WALK_ERR);
649     }

651     tcp_kaddr = (uintptr_t)connp->conn_tcp;
652     if (mdb_vread(&tcps, sizeof (tcp_t), tcp_kaddr) == -1) {
653         mdb_warn("failed to read tcp_t at %p", tcp_kaddr);
654         return (WALK_ERR);
655     }

```

```

657     tcp = &tcps;
658     connp->conn_tcp = tcp;
659     tcp->tcp_connp = connp;

661     if (!(opts & NETSTAT_ALL) || net_tcp_active(tcp)) ||
662         (af == AF_INET && !net_tcp_ipv4(tcp)) ||
663         (af == AF_INET6 && !net_tcp_ipv6(tcp))) {
664         return (WALK_NEXT);
665     }

667     mdb_printf("%0?p %2i ", tcp_kaddr, tcp->tcp_state);
668     if (af == AF_INET) {
669         net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
670         mdb_printf(" ");
671         net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
672     } else if (af == AF_INET6) {
673         net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
674         mdb_printf(" ");
675         net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
676     }
677     mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
678     mdb_printf(" %4i\n", connp->conn_zoneid);
679     if (opts & NETSTAT_VERBOSE)
680         netstat_tcp_verbose_pr(tcp);

682     return (WALK_NEXT);
683 }

685 /*ARGSUSED*/
686 static int
687 netstat_udp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
688 {
689     netstat_cb_data_t *ncb = cb_data;
690     uint_t opts = ncb->opts;
691     int af = ncb->af;
692     udp_t udp;
693     conn_t *connp = &ncb->conn;
694     char *state;

696     if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
697         mdb_warn("failed to read conn_t at %p", kaddr);
698         return (WALK_ERR);
699     }

701     if (mdb_vread(&udp, sizeof (udp_t),
702         (uintptr_t)connp->conn_udp) == -1) {
703         mdb_warn("failed to read conn_udp at %p",
704             (uintptr_t)connp->conn_udp);
705         return (WALK_ERR);
706     }

708     connp->conn_udp = &udp;
709     udp.udp_connp = connp;

711     if (!(opts & NETSTAT_ALL) || net_udp_active(&udp)) ||
712         (af == AF_INET && !net_udp_ipv4(&udp)) ||
713         (af == AF_INET6 && !net_udp_ipv6(&udp))) {
714         return (WALK_NEXT);
715     }

717     if (udp.udp_state == TS_UNBND)
718         state = "UNBOUND";
719     else if (udp.udp_state == TS_IDLE)
720         state = "IDLE";
721     else if (udp.udp_state == TS_DATA_XFER)

```

```

722         state = "CONNECTED";
723     else
724         state = "UNKNOWN";

726     mdb_printf("%0?p %10s ", (uintptr_t)connp->conn_udp, state);
727     if (af == AF_INET) {
728         net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
729         mdb_printf(" ");
730         net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
731     } else if (af == AF_INET6) {
732         net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
733         mdb_printf(" ");
734         net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
735     }
736     mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
737     mdb_printf(" %4i\n", connp->conn_zoneid);

739     return (WALK_NEXT);
740 }

742 /*ARGSUSED*/
743 static int
744 netstat_icmp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
745 {
746     netstat_cb_data_t *ncb = cb_data;
747     int af = ncb->af;
748     icmp_t icmp;
749     conn_t *connp = &ncb->conn;
750     char *state;

752     if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
753         mdb_warn("failed to read conn_t at %p", kaddr);
754         return (WALK_ERR);
755     }

757     if (mdb_vread(&icmp, sizeof (icmp_t),
758         (uintptr_t)connp->conn_icmp) == -1) {
759         mdb_warn("failed to read conn_icmp at %p",
760             (uintptr_t)connp->conn_icmp);
761         return (WALK_ERR);
762     }

764     connp->conn_icmp = &icmp;
765     icmp.icmp_connp = connp;

767     if ((af == AF_INET && connp->conn_ipversion != IPV4_VERSION) ||
768         (af == AF_INET6 && connp->conn_ipversion != IPV6_VERSION)) {
769         return (WALK_NEXT);
770     }

772     if (icmp.icmp_state == TS_UNBND)
773         state = "UNBOUND";
774     else if (icmp.icmp_state == TS_IDLE)
775         state = "IDLE";
776     else if (icmp.icmp_state == TS_DATA_XFER)
777         state = "CONNECTED";
778     else
779         state = "UNKNOWN";

781     mdb_printf("%0?p %10s ", (uintptr_t)connp->conn_icmp, state);
782     if (af == AF_INET) {
783         net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
784         mdb_printf(" ");
785         net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
786     } else if (af == AF_INET6) {
787         net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);

```



```

788         mdb_printf(" ");
789         net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
790     }
791     mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
792     mdb_printf(" %4i\n", connp->conn_zoneid);
793
794     return (WALK_NEXT);
795 }
796
797 static void
798 netstat_dccp_verbose_pr(const dccp_t *dccp)
799 {
800     /* XXX:DCCP
801     mdb_printf("      %5i %08x %08x %5i %08x %08x %5li %5i\n",
802         tcp->tcp_swnd, tcp->tcp_snxt, tcp->tcp_suna, tcp->tcp_rwnd,
803         tcp->tcp_rack, tcp->tcp_rnxt, tcp->tcp_rto, tcp->tcp_mss);
804     */
805 }
806
807 /*ARGSUSED*/
808 static int
809 netstat_dccp_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
810 {
811     netstat_cb_data_t *ncb = cb_data;
812     uint_t opts = ncb->opts;
813     int af = ncb->af;
814     uintptr_t dccp_kaddr;
815     conn_t *connp = &ncb->conn;
816     dccp_t dccps, *dccp;
817
818     if (mdb_vread(connp, sizeof (conn_t), kaddr) == -1) {
819         mdb_warn("failed to read conn_t at %p", kaddr);
820         return (WALK_ERR);
821     }
822
823     dccp_kaddr = (uintptr_t)connp->conn_dccp;
824     if (mdb_vread(&dccps, sizeof (dccp_t), dccp_kaddr) == -1) {
825         mdb_warn("failed to read tcp_t at %p", dccp_kaddr);
826         return (WALK_ERR);
827     }
828
829     dccp = &dccps;
830     connp->conn_dccp = dccp;
831     dccp->dccp_connp = connp;
832
833     if (((opts & NETSTAT_ALL) || net_dccp_active(dccp)) ||
834         (af == AF_INET && !net_dccp_ipv4(dccp)) ||
835         (af == AF_INET6 && !net_dccp_ipv6(dccp))) {
836         return (WALK_NEXT);
837     }
838
839     mdb_printf("%0?p %2i ", dccp_kaddr, dccp->dccp_state);
840     if (af == AF_INET) {
841         net_ipv4addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
842         mdb_printf(" ");
843         net_ipv4addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
844     } else if (af == AF_INET6) {
845         net_ipv6addrport_pr(&connp->conn_laddr_v6, connp->conn_lport);
846         mdb_printf(" ");
847         net_ipv6addrport_pr(&connp->conn_faddr_v6, connp->conn_fport);
848     }
849     mdb_printf(" %5i", ns_to_stackid((uintptr_t)connp->conn_netstack));
850     mdb_printf(" %4i\n", connp->conn_zoneid);
851     if (opts & NETSTAT_VERBOSE)
852         netstat_dccp_verbose_pr(dccp);

```

```

854         return (WALK_NEXT);
855     }
856
857 #endif /* ! codereview */
858 /*
859  * print the address of a unix domain socket
860  * so is the address of a AF_UNIX struct sonode in mdb's address space
861  * soa is the address of the struct soaddr to print
862  * returns 0 on success, -1 otherwise
863  */
864 static int
865 netstat_unix_name_pr(const struct sotpi_sonode *st, const struct soaddr *soa)
866 {
867     const struct sonode *so = &st->st_sonode;
868     const char none[] = " (none)";
869
870     if ((so->so_state & SS_ISBOUND) && (soa->soa_len != 0)) {
871         if (st->st_info.sti_faddr_noxlate) {
872             mdb_printf("%-14s ", " (socketpair)");
873         } else {
874             if (soa->soa_len > sizeof (sa_family_t)) {
875                 char addr[MAXPATHLEN + 1];
876
877                 if (mdb_readstr(addr, sizeof (addr),
878                     (uintptr_t)&soa->soa_sa->sa_data) == -1) {
879                     mdb_warn("failed to read unix address "
880                         "at %p", &soa->soa_sa->sa_data);
881                     return (-1);
882                 }
883
884                 mdb_printf("%-14s ", addr);
885             } else {
886                 mdb_printf("%-14s ", none);
887             }
888         }
889     } else {
890         mdb_printf("%-14s ", none);
891     }
892
893     return (0);
894 }
895
896 /* based on sockfs_snapshot */
897 /*ARGSUSED*/
898 static int
899 netstat_unix_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
900 {
901     const struct sotpi_sonode *st = walk_data;
902     const struct sonode *so = &st->st_sonode;
903     const struct sotpi_info *sti = &st->st_info;
904
905     if (so->so_count == 0)
906         return (WALK_NEXT);
907
908     if (so->so_family != AF_UNIX) {
909         mdb_warn("sonode of family %hi at %p\n", so->so_family, kaddr);
910         return (WALK_ERR);
911     }
912
913     mdb_printf("%-?p ", kaddr);
914
915     switch (sti->sti_serv_type) {
916     case T_CLTS:
917         mdb_printf("%-10s ", "dgram");

```

```

920         break;
921     case T_COTS:
922         mdb_printf("%-10s ", "stream");
923         break;
924     case T_COTS_ORD:
925         mdb_printf("%-10s ", "stream-ord");
926         break;
927     default:
928         mdb_printf("%-10i ", sti->sti_serv_type);
929     }

931     if ((so->so_state & SS_ISBOUND) &&
932         (sti->sti_ux_laddr.soua_magic == SOU_MAGIC_EXPLICIT)) {
933         mdb_printf("%0?p ", sti->sti_ux_laddr.soua_vp);
934     } else {
935         mdb_printf("%0?p ", NULL);
936     }

938     if ((so->so_state & SS_ISCONNECTED) &&
939         (sti->sti_ux_faddr.soua_magic == SOU_MAGIC_EXPLICIT)) {
940         mdb_printf("%0?p ", sti->sti_ux_faddr.soua_vp);
941     } else {
942         mdb_printf("%0?p ", NULL);
943     }

945     if (netstat_unix_name_pr(st, &sti->sti_laddr) == -1)
946         return (WALK_ERR);

948     if (netstat_unix_name_pr(st, &sti->sti_faddr) == -1)
949         return (WALK_ERR);

951     mdb_printf("%4i\n", so->so_zoneid);

953     return (WALK_NEXT);
954 }

956 static void
957 netstat_tcp_verbose_header_pr(void)
958 {
959     mdb_printf("      <u>%-5s %-8s %-8s %-5s %-8s %-8s %5s %5s</u>\n",
960         "Swind", "Snext", "Suna", "Rwind", "Rack", "Rnext", "Rto", "Mss");
961 }

963 static void
964 get_ifname(const ire_t *ire, char *intf)
965 {
966     ill_t ill;

968     *intf = '\0';
969     if (ire->ire_ill != NULL) {
970         if (mdb_vread(&ill, sizeof(ill),
971             (uintptr_t)ire->ire_ill) == -1)
972             return;
973         (void) mdb_readstr(intf, MIN(LIFNAMSIZ, ill.ill_name_length),
974             (uintptr_t)ill.ill_name);
975     }
976 }

978 const in6_addr_t ipv6_all_ones =
979     { 0xffffffffU, 0xffffffffU, 0xffffffffU, 0xffffffffU };

981 static void
982 get_ireflags(const ire_t *ire, char *flags)
983 {
984     (void) strcpy(flags, "U");
985     /* RTF_INDIRECT wins over RTF_GATEWAY - don't display both */

```

```

986     if (ire->ire_flags & RTF_INDIRECT)
987         (void) strcat(flags, "I");
988     else if (ire->ire_type & IRE_OFFLINK)
989         (void) strcat(flags, "G");

991     /* IRE_IF_CLONE wins over RTF_HOST - don't display both */
992     if (ire->ire_type & IRE_IF_CLONE)
993         (void) strcat(flags, "C");
994     else if (ire->ire_ipversion == IPV4_VERSION) {
995         if (ire->ire_mask == IP_HOST_MASK)
996             (void) strcat(flags, "H");
997     } else {
998         if (IN6_ARE_ADDR_EQUAL(&ire->ire_mask_v6, &ipv6_all_ones))
999             (void) strcat(flags, "H");
1000     }

1002     if (ire->ire_flags & RTF_DYNAMIC)
1003         (void) strcat(flags, "D");
1004     if (ire->ire_type == IRE_BROADCAST)
1005         (void) strcat(flags, "b");
1006     if (ire->ire_type == IRE_MULTICAST)
1007         (void) strcat(flags, "m");
1008     if (ire->ire_type == IRE_LOCAL)
1009         (void) strcat(flags, "L");
1010     if (ire->ire_type == IRE_NOROUTE)
1011         (void) strcat(flags, "N");
1012     if (ire->ire_flags & RTF_MULTIRT)
1013         (void) strcat(flags, "M");
1014     if (ire->ire_flags & RTF_SETSRC)
1015         (void) strcat(flags, "S");
1016     if (ire->ire_flags & RTF_REJECT)
1017         (void) strcat(flags, "R");
1018     if (ire->ire_flags & RTF_BLACKHOLE)
1019         (void) strcat(flags, "B");
1020 }

1022 static int
1023 netstat_irev4_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
1024 {
1025     const ire_t *ire = walk_data;
1026     uint_t *opts = cb_data;
1027     ipaddr_t gate;
1028     char flags[10], intf[LIFNAMSIZ + 1];

1030     if (ire->ire_ipversion != IPV4_VERSION)
1031         return (WALK_NEXT);

1033     /* Skip certain IRES by default */
1034     if (!(*opts & NETSTAT_ALL) &&
1035         (ire->ire_type &
1036             (IRE_BROADCAST|IRE_LOCAL|IRE_MULTICAST|IRE_NOROUTE|IRE_IF_CLONE)))
1037         return (WALK_NEXT);

1039     if (*opts & NETSTAT_FIRST) {
1040         *opts &= ~NETSTAT_FIRST;
1041         mdb_printf("<u>%s Table: IPv4</u>\n",
1042             (*opts & NETSTAT_VERBOSE) ? "IRE" : "Routing");
1043         if (*opts & NETSTAT_VERBOSE) {
1044             mdb_printf("<u>%-?s %-?s %-?s %-?s Device Mxfrg Rtt  "
1045                 " Ref Flg Out  In/Fwd</u>\n",
1046                 "Address", ADDR_V4_WIDTH, "Destination",
1047                 ADDR_V4_WIDTH, "Mask", ADDR_V4_WIDTH, "Gateway");
1048         } else {
1049             mdb_printf("<u>%-?s %-?s %-?s Flags Ref  Use  "
1050                 "Interface</u>\n",
1051                 "Address", ADDR_V4_WIDTH, "Destination",

```

```

1052         ADDR_V4_WIDTH, "Gateway");
1053     }
1054 }
1055
1056 gate = ire->ire_gateway_addr;
1057
1058 get_ireflags(ire, flags);
1059
1060 get_ifname(ire, intf);
1061
1062 if (*opts & NETSTAT_VERBOSE) {
1063     mdb_printf("%?p %-*I %-*I %-*I %-6s %5u%c %4u %3u %-3s %5u "
1064               "%u\n", kaddr, ADDR_V4_WIDTH, ire->ire_addr, ADDR_V4_WIDTH,
1065               ire->ire_mask, ADDR_V4_WIDTH, gate, intf,
1066               0, ' ',
1067               ire->ire_metrics.iulp_rtt, ire->ire_refcnt, flags,
1068               ire->ire_ob_pkt_count, ire->ire_ib_pkt_count);
1069 } else {
1070     mdb_printf("%?p %-*I %-*I %-5s %4u %5u %s\n", kaddr,
1071               ADDR_V4_WIDTH, ire->ire_addr, ADDR_V4_WIDTH, gate, flags,
1072               ire->ire_refcnt,
1073               ire->ire_ob_pkt_count + ire->ire_ib_pkt_count, intf);
1074 }
1075
1076 return (WALK_NEXT);
1077 }
1078
1079 int
1080 ip_mask_to_plen_v6(const in6_addr_t *v6mask)
1081 {
1082     int plen;
1083     int i;
1084     uint32_t val;
1085
1086     for (i = 3; i >= 0; i--)
1087         if (v6mask->s6_addr32[i] != 0)
1088             break;
1089     if (i < 0)
1090         return (0);
1091     plen = 32 + 32 * i;
1092     val = v6mask->s6_addr32[i];
1093     while (!(val & 1)) {
1094         val >>= 1;
1095         plen--;
1096     }
1097
1098     return (plen);
1099 }
1100
1101 static int
1102 netstat_irev6_cb(uintptr_t kaddr, const void *walk_data, void *cb_data)
1103 {
1104     const ire_t *ire = walk_data;
1105     uint_t *opts = cb_data;
1106     const in6_addr_t *gatep;
1107     char deststr[ADDR_V6_WIDTH + 5];
1108     char flags[10], intf[LIFNAMSIZ + 1];
1109     int masklen;
1110
1111     if (ire->ire_ipversion != IPV6_VERSION)
1112         return (WALK_NEXT);
1113
1114     /* Skip certain IRES by default */
1115     if (!(opts & NETSTAT_ALL) &&
1116         (ire->ire_type &
1117          (IRE_BROADCAST|IRE_LOCAL|IRE_MULTICAST|IRE_NOROUTE|IRE_IF_CLONE)))

```

```

1118     return (WALK_NEXT);
1119
1120     if (*opts & NETSTAT_FIRST) {
1121         *opts &= ~NETSTAT_FIRST;
1122         mdb_printf("\n%<u>%s Table: IPv6%</u>\n",
1123                   (*opts & NETSTAT_VERBOSE) ? "IRE" : "Routing");
1124         if (*opts & NETSTAT_VERBOSE) {
1125             mdb_printf("%<u>%-?s %-*s %-*s If PMTU Rtt Ref "
1126                       "Flags Out In/Fwd%</u>\n",
1127                       "Address", ADDR_V6_WIDTH+4, "Destination/Mask",
1128                       ADDR_V6_WIDTH, "Gateway");
1129         } else {
1130             mdb_printf("%<u>%-?s %-*s %-*s Flags Ref Use If "
1131                       "%</u>\n",
1132                       "Address", ADDR_V6_WIDTH+4, "Destination/Mask",
1133                       ADDR_V6_WIDTH, "Gateway");
1134         }
1135     }
1136
1137     gatep = &ire->ire_gateway_addr_v6;
1138
1139     masklen = ip_mask_to_plen_v6(&ire->ire_mask_v6);
1140     (void) mdb_snprintf(deststr, sizeof(deststr), "%N/%d",
1141                       &ire->ire_addr_v6, masklen);
1142
1143     get_ireflags(ire, flags);
1144
1145     get_ifname(ire, intf);
1146
1147     if (*opts & NETSTAT_VERBOSE) {
1148         mdb_printf("%?p %-*s %-*N %-5s %5u%c %5u %3u %-5s %6u %u\n",
1149                   kaddr, ADDR_V6_WIDTH+4, deststr, ADDR_V6_WIDTH, gatep,
1150                   intf, 0, ' ',
1151                   ire->ire_metrics.iulp_rtt, ire->ire_refcnt,
1152                   flags, ire->ire_ob_pkt_count, ire->ire_ib_pkt_count);
1153     } else {
1154         mdb_printf("%?p %-*s %-*N %-5s %3u %6u %s\n", kaddr,
1155                   ADDR_V6_WIDTH+4, deststr, ADDR_V6_WIDTH, gatep, flags,
1156                   ire->ire_refcnt,
1157                   ire->ire_ob_pkt_count + ire->ire_ib_pkt_count, intf);
1158     }
1159
1160     return (WALK_NEXT);
1161 }
1162
1163 static void
1164 netstat_header_v4(int proto)
1165 {
1166     if (proto == IPPROTO_TCP)
1167         mdb_printf("%<u>%-?s ", "TCPv4");
1168     else if (proto == IPPROTO_UDP)
1169         mdb_printf("%<u>%-?s ", "UDPv4");
1170     else if (proto == IPPROTO_ICMP)
1171         mdb_printf("%<u>%-?s ", "ICMPv4");
1172     mdb_printf("State %6s%*s %6s%*s %-5s %-4s%</u>\n",
1173               "", ADDR_V4_WIDTH, "Local Address",
1174               "", ADDR_V4_WIDTH, "Remote Address", "Stack", "Zone");
1175 }
1176
1177 static void
1178 netstat_header_v6(int proto)
1179 {
1180     if (proto == IPPROTO_TCP)
1181         mdb_printf("%<u>%-?s ", "TCPv6");
1182     else if (proto == IPPROTO_UDP)
1183         mdb_printf("%<u>%-?s ", "UDPv6");

```

```

1184     else if (proto == IPPROTO_ICMP)
1185         mdb_printf("%<u>%-?s ", "ICMPv6");
1186     mdb_printf("State %6s*s %6s*s %-5s %-4s%</u>\n",
1187         "", ADDR_V6_WIDTH, "Local Address",
1188         "", ADDR_V6_WIDTH, "Remote Address", "Stack", "Zone");
1189 }

1191 static int
1192 netstat_print_conn(const char *cache, int proto, mdb_walk_cb_t cbfunc,
1193     void *cbdata)
1194 {
1195     netstat_cb_data_t *ncb = cbdata;

1197     if ((ncb->opts & NETSTAT_VERBOSE) && proto == IPPROTO_TCP)
1198         netstat_tcp_verbose_header_pr();
1199     if (mdb_walk(cache, cbfunc, cbdata) == -1) {
1200         mdb_warn("failed to walk %s", cache);
1201         return (DCMD_ERR);
1202     }
1203     return (DCMD_OK);
1204 }

1206 static int
1207 netstat_print_common(const char *cache, int proto, mdb_walk_cb_t cbfunc,
1208     void *cbdata)
1209 {
1210     netstat_cb_data_t *ncb = cbdata;
1211     int af = ncb->af;
1212     int status = DCMD_OK;

1214     if (af != AF_INET6) {
1215         ncb->af = AF_INET;
1216         netstat_header_v4(proto);
1217         status = netstat_print_conn(cache, proto, cbfunc, cbdata);
1218     }
1219     if (status == DCMD_OK && af != AF_INET) {
1220         ncb->af = AF_INET6;
1221         netstat_header_v6(proto);
1222         status = netstat_print_conn(cache, proto, cbfunc, cbdata);
1223     }
1224     ncb->af = af;
1225     return (status);
1226 }

1228 /*ARGSUSED*/
1229 int
1230 netstat(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1231 {
1232     uint_t opts = 0;
1233     const char *optf = NULL;
1234     const char *optP = NULL;
1235     netstat_cb_data_t *cbdata;
1236     int status;
1237     int af = 0;

1239     if (mdb_getopts(argc, argv,
1240         'a', MDB_OPT_SETBITS, NETSTAT_ALL, &opts,
1241         'f', MDB_OPT_STR, &optf,
1242         'P', MDB_OPT_STR, &optP,
1243         'r', MDB_OPT_SETBITS, NETSTAT_ROUTE, &opts,
1244         'v', MDB_OPT_SETBITS, NETSTAT_VERBOSE, &opts,
1245         NULL) != argc)
1246         return (DCMD_USAGE);

1248     if (optP != NULL) {
1249         if ((strcmp("tcp", optP) != 0) && (strcmp("udp", optP) != 0) &&

```

```

1250         (strcmp("icmp", optP) != 0))
1251             return (DCMD_USAGE);
1252         if (opts & NETSTAT_ROUTE)
1253             return (DCMD_USAGE);
1254     }

1256     if (optf == NULL)
1257         opts |= NETSTAT_V4 | NETSTAT_V6 | NETSTAT_UNIX;
1258     else if (strcmp("inet", optf) == 0)
1259         opts |= NETSTAT_V4;
1260     else if (strcmp("inet6", optf) == 0)
1261         opts |= NETSTAT_V6;
1262     else if (strcmp("unix", optf) == 0)
1263         opts |= NETSTAT_UNIX;
1264     else
1265         return (DCMD_USAGE);

1267     if (opts & NETSTAT_ROUTE) {
1268         if (!(opts & (NETSTAT_V4|NETSTAT_V6)))
1269             return (DCMD_USAGE);
1270         if (opts & NETSTAT_V4) {
1271             opts |= NETSTAT_FIRST;
1272             if (mdb_walk("ip'ire", netstat_irev4_cb, &opts) == -1) {
1273                 mdb_warn("failed to walk ip'ire");
1274                 return (DCMD_ERR);
1275             }
1276         }
1277         if (opts & NETSTAT_V6) {
1278             opts |= NETSTAT_FIRST;
1279             if (mdb_walk("ip'ire", netstat_irev6_cb, &opts) == -1) {
1280                 mdb_warn("failed to walk ip'ire");
1281                 return (DCMD_ERR);
1282             }
1283         }
1284         return (DCMD_OK);
1285     }

1287     if ((opts & NETSTAT_UNIX) && (optP == NULL)) {
1288         /* Print Unix Domain Sockets */
1289         mdb_printf("%<u>%-?s %-10s %-?s %-?s %-14s %-14s %s%</u>\n",
1290             "AF_UNIX", "Type", "Vnode", "Conn", "Local Addr",
1291             "Remote Addr", "Zone");

1293         if (mdb_walk("genunix'sonode", netstat_unix_cb, NULL) == -1) {
1294             mdb_warn("failed to walk genunix'sonode");
1295             return (DCMD_ERR);
1296         }
1297         if (!(opts & (NETSTAT_V4 | NETSTAT_V6)))
1298             return (DCMD_OK);
1299     }

1301     cbdata = mdb_alloc(sizeof (netstat_cb_data_t), UM_SLEEP);
1302     cbdata->opts = opts;
1303     if ((optf != NULL) && (opts & NETSTAT_V4))
1304         af = AF_INET;
1305     else if ((optf != NULL) && (opts & NETSTAT_V6))
1306         af = AF_INET6;

1308     cbdata->af = af;
1309     if ((optP == NULL) || (strcmp("tcp", optP) == 0)) {
1310         status = netstat_print_common("tcp_conn_cache", IPPROTO_TCP,
1311             netstat_tcp_cb, cbdata);
1312         if (status != DCMD_OK)
1313             goto out;
1314     }

```

```

1316     if ((optP == NULL) || (strcmp("udp", optP) == 0)) {
1317         status = netstat_print_common("udp_conn_cache", IPPROTO_UDP,
1318             netstat_udp_cb, cbdata);
1319         if (status != DCMD_OK)
1320             goto out;
1321     }
1322
1323     if ((optP == NULL) || (strcmp("icmp", optP) == 0)) {
1324         status = netstat_print_common("rawip_conn_cache", IPPROTO_ICMP,
1325             netstat_icmp_cb, cbdata);
1326         if (status != DCMD_OK)
1327             goto out;
1328     }
1329
1330     if ((optP == NULL) || (strcmp("dccp", optP) == 0)) {
1331         status = netstat_print_common("dccp_conn_cache", IPPROTO_DCCP,
1332             netstat_dccp_cb, cbdata);
1333         if (status != DCMD_OK)
1334             goto out;
1335     }
1336 #endif /* ! codereview */
1337 out:
1338     mdb_free(cbdata, sizeof (netstat_cb_data_t));
1339     return (status);
1340 }
1341
1342 /*
1343  * "::dladm show-bridge" support
1344  */
1345 typedef struct {
1346     uint_t opt_l;
1347     uint_t opt_f;
1348     uint_t opt_t;
1349     const char *name;
1350     clock_t lbolt;
1351     boolean_t found;
1352     uint_t nlinks;
1353     uint_t nfwd;
1354 } show_bridge_args_t;
1355
1356 /*
1357  * These structures are kept inside the 'args' for allocation reasons.
1358  * They're all large data structures (over 1K), and may cause the stack
1359  * to explode. mdb and kmdb will fail in these cases, and thus we
1360  * allocate them from the heap.
1361  */
1362 trill_inst_t ti;
1363 bridge_link_t bl;
1364 mac_impl_t mi;
1365
1366 static void
1367 show_vlans(const uint8_t *vlans)
1368 {
1369     int i, bit;
1370     uint8_t val;
1371     int rstart = -1, rnext = -1;
1372
1373     for (i = 0; i < BRIDGE_VLAN_ARR_SIZE; i++) {
1374         val = vlans[i];
1375         if (i == 0)
1376             val &= ~1;
1377         while ((bit = mdb_ffs(val)) != 0) {
1378             bit--;
1379             val &= ~(1 << bit);
1380             bit += i * sizeof (*vlans) * NBBY;
1381             if (bit != rnext) {

```

```

1382         if (rnext != -1 && rstart + 1 != rnext)
1383             mdb_printf("-%d", rnext - 1);
1384         if (rstart != -1)
1385             mdb_printf(",");
1386         mdb_printf("%d", bit);
1387         rstart = bit;
1388     }
1389     rnext = bit + 1;
1390 }
1391 }
1392 if (rnext != -1 && rstart + 1 != rnext)
1393     mdb_printf("-%d", rnext - 1);
1394     mdb_printf("\n");
1395 }
1396
1397 /*
1398  * This callback is invoked by a walk of the links attached to a bridge. If
1399  * we're showing link details, then they're printed here. If not, then we just
1400  * count up the links for the bridge summary.
1401  */
1402 static int
1403 do_bridge_links(uintptr_t addr, const void *data, void *ptr)
1404 {
1405     show_bridge_args_t *args = ptr;
1406     const bridge_link_t *blp = data;
1407     char macaddr[ETHERADDRL * 3];
1408     const char *name;
1409
1410     args->nlinks++;
1411
1412     if (!args->opt_l)
1413         return (WALK_NEXT);
1414
1415     if (mdb_vread(&args->mi, sizeof (args->mi),
1416         (uintptr_t)blp->bl_mh) == -1) {
1417         mdb_warn("cannot read mac data at %p", blp->bl_mh);
1418         name = "?";
1419     } else {
1420         name = args->mi.mi_name;
1421     }
1422
1423     mdb_mac_addr(blp->bl_local_mac, ETHERADDRL, macaddr,
1424         sizeof (macaddr));
1425
1426     mdb_printf("%-?p %-16s %-17s %03X %-4d ", addr, name, macaddr,
1427         blp->bl_flags, blp->bl_pvid);
1428
1429     if (blp->bl_trilldata == NULL) {
1430         switch (blp->bl_state) {
1431             case BLS_BLOCKLISTEN:
1432                 name = "BLOCK";
1433                 break;
1434             case BLS_LEARNING:
1435                 name = "LEARN";
1436                 break;
1437             case BLS_FORWARDING:
1438                 name = "FWD";
1439                 break;
1440             default:
1441                 name = "?";
1442         }
1443         mdb_printf("%-5s ", name);
1444         show_vlans(blp->bl_vlans);
1445     } else {
1446         show_vlans(blp->bl_afs);
1447     }

```

```

1449     return (WALK_NEXT);
1450 }

1452 /*
1453  * It seems a shame to duplicate this code, but merging it with the link
1454  * printing code above is more trouble than it would be worth.
1455  */
1456 static void
1457 print_link_name(show_bridge_args_t *args, uintptr_t addr, char sep)
1458 {
1459     const char *name;

1461     if (mdb_vread(&args->bl, sizeof (args->bl), addr) == -1) {
1462         mdb_warn("cannot read bridge link at %p", addr);
1463         return;
1464     }

1466     if (mdb_vread(&args->mi, sizeof (args->mi),
1467         (uintptr_t)args->bl.bl_mh) == -1) {
1468         name = "?";
1469     } else {
1470         name = args->mi.mi_name;
1471     }

1473     mdb_printf("%s%c", name, sep);
1474 }

1476 static int
1477 do_bridge_fwd(uintptr_t addr, const void *data, void *ptr)
1478 {
1479     show_bridge_args_t *args = ptr;
1480     const bridge_fwd_t *bfp = data;
1481     char macaddr[ETHERADDRL * 3];
1482     int i;
1483 #define MAX_FWD_LINKS 16
1484     bridge_link_t *links[MAX_FWD_LINKS];
1485     uint_t nlinks;

1487     args->n fwd++;

1489     if (!args->opt_f)
1490         return (WALK_NEXT);

1492     if ((nlinks = bfp->bf_nlinks) > MAX_FWD_LINKS)
1493         nlinks = MAX_FWD_LINKS;

1495     if (mdb_vread(links, sizeof (links[0]) * nlinks,
1496         (uintptr_t)bfp->bf_links) == -1) {
1497         mdb_warn("cannot read bridge forwarding links at %p",
1498             bfp->bf_links);
1499         return (WALK_ERR);
1500     }

1502     mdb_mac_addr(bfp->bf_dest, ETHERADDRL, macaddr, sizeof (macaddr));

1504     mdb_printf("%-?p %-17s ", addr, macaddr);
1505     if (bfp->bf_flags & BFF_LOCALADDR)
1506         mdb_printf("%-7s", "[self]");
1507     else
1508         mdb_printf("t-%5d", args->lbolt - bfp->bf_lastheard);
1509     mdb_printf(" %-7u ", bfp->bf_refs);

1511     if (bfp->bf_trill_nick != 0) {
1512         mdb_printf("%d\n", bfp->bf_trill_nick);
1513     } else {

```

```

1514         for (i = 0; i < bfp->bf_nlinks; i++) {
1515             print_link_name(args, (uintptr_t)links[i],
1516                 i == bfp->bf_nlinks - 1 ? '\n' : ' ');
1517         }
1518     }

1520     return (WALK_NEXT);
1521 }

1523 static int
1524 do_show_bridge(uintptr_t addr, const void *data, void *ptr)
1525 {
1526     show_bridge_args_t *args = ptr;
1527     bridge_inst_t bi;
1528     const bridge_inst_t *bip;
1529     trill_node_t tn;
1530     trill_sock_t tsp;
1531     trill_nickname_t tni;
1532     char bname[MAXLINKNAMELEN];
1533     char macaddr[ETHERADDRL * 3];
1534     char *cp;
1535     uint_t nnicks;
1536     int i;

1538     if (data != NULL) {
1539         bip = data;
1540     } else {
1541         if (mdb_vread(&bi, sizeof (bi), addr) == -1) {
1542             mdb_warn("cannot read bridge instance at %p", addr);
1543             return (WALK_ERR);
1544         }
1545         bip = &bi;
1546     }

1548     (void) strncpy(bname, bip->bi_name, sizeof (bname) - 1);
1549     bname[MAXLINKNAMELEN - 1] = '\0';
1550     cp = bname + strlen(bname);
1551     if (cp > bname && cp[-1] == '0')
1552         cp[-1] = '\0';

1554     if (args->name != NULL && strcmp(args->name, bname) != 0)
1555         return (WALK_NEXT);

1557     args->found = B_TRUE;
1558     args->nlinks = args->n fwd = 0;

1560     if (args->opt_l) {
1561         mdb_printf("%-?s %-16s %-17s %3s %-4s ", "ADDR", "LINK",
1562             "MAC-ADDR", "FLG", "PVID");
1563         if (bip->bi_trilldata == NULL)
1564             mdb_printf("%-5s %s\n", "STATE", "VLANS");
1565         else
1566             mdb_printf("%s\n", "FWD-VLANS");
1567     }

1569     if (!args->opt_f && !args->opt_t &&
1570         mdb_pwalk("list", do_bridge_links, args,
1571             addr + offsetof(bridge_inst_t, bi_links)) != DCMD_OK)
1572         return (WALK_ERR);

1574     if (args->opt_f)
1575         mdb_printf("%-?s %-17s %-7s %-7s %s\n", "ADDR", "DEST", "TIME",
1576             "REFS", "OUTPUT");

1578     if (!args->opt_l && !args->opt_t &&
1579         mdb_pwalk("avl", do_bridge_fwd, args,

```

```

1580     addr + offsetof(bridge_inst_t, bi_fwd)) != DCMD_OK)
1581         return (WALK_ERR);

1583     nnicks = 0;
1584     if (bip->bi_trilldata != NULL && !args->opt_l && !args->opt_f) {
1585         if (mdb_vread(&args->ti, sizeof (args->ti),
1586             (uintptr_t)bip->bi_trilldata) == -1) {
1587             mdb_warn("cannot read trill instance at %p",
1588                 bip->bi_trilldata);
1589             return (WALK_ERR);
1590         }
1591         if (args->opt_t)
1592             mdb_printf("%-?s %-5s %-17s %s\n", "ADDR",
1593                 "NICK", "NEXT-HOP", "LINK");
1594         for (i = 0; i < RBRIDGE_NICKNAME_MAX; i++) {
1595             if (args->ti.ti_nodes[i] == NULL)
1596                 continue;
1597             if (args->opt_t) {
1598                 if (mdb_vread(&tn, sizeof (tn),
1599                     (uintptr_t)args->ti.ti_nodes[i]) == -1) {
1600                     mdb_warn("cannot read trill node %d at "
1601                         "%p", i, args->ti.ti_nodes[i]);
1602                     return (WALK_ERR);
1603                 }
1604                 if (mdb_vread(&tni, sizeof (tni),
1605                     (uintptr_t)tn.tn_ni) == -1) {
1606                     mdb_warn("cannot read trill node info "
1607                         "%d at %p", i, tn.tn_ni);
1608                     return (WALK_ERR);
1609                 }
1610                 mdb_mac_addr(tni.tni_adjsnpa, ETHERADDRL,
1611                     macaddr, sizeof (macaddr));
1612                 if (tni.tni_nick == args->ti.ti_nick) {
1613                     (void) strcpy(macaddr, "[self]");
1614                 }
1615                 mdb_printf("%-?p %-5u %-17s ",
1616                     args->ti.ti_nodes[i], tni.tni_nick,
1617                     macaddr);
1618                 if (tn.tn_tsp != NULL) {
1619                     if (mdb_vread(&tsp, sizeof (tsp),
1620                         (uintptr_t)tn.tn_tsp) == -1) {
1621                         mdb_warn("cannot read trill "
1622                             "socket info at %p",
1623                             tn.tn_tsp);
1624                         return (WALK_ERR);
1625                     }
1626                     if (tsp.ts_link != NULL) {
1627                         print_link_name(args,
1628                             (uintptr_t)tsp.ts_link,
1629                             '\n');
1630                         continue;
1631                     }
1632                 }
1633                 mdb_printf("--\n");
1634             } else {
1635                 nnicks++;
1636             }
1637         }
1638     } else {
1639         if (args->opt_t)
1640             mdb_printf("bridge is not running TRILL\n");
1641     }

1643     if (!args->opt_l && !args->opt_f && !args->opt_t) {
1644         mdb_printf("%-?p %-7s %-16s %-7u %-7u", addr,
1645             bip->bi_trilldata == NULL ? "stp" : "trill", bname,

```

```

1646         args->nlinks, args->nfwd);
1647         if (bip->bi_trilldata != NULL)
1648             mdb_printf(" %-7u %u\n", nnicks, args->ti.ti_nick);
1649         else
1650             mdb_printf(" %-7s %s\n", "--", "--");
1651     }
1652     return (WALK_NEXT);
1653 }

1655 static int
1656 dladm_show_bridge(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1657 {
1658     show_bridge_args_t *args;
1659     GElf_Sym sym;
1660     int i;

1662     args = mdb_zalloc(sizeof (*args), UM_SLEEP);

1664     i = mdb_getopts(argc, argv,
1665         'l', MDB_OPT_SETBITS, 1, &args->opt_l,
1666         'f', MDB_OPT_SETBITS, 1, &args->opt_f,
1667         't', MDB_OPT_SETBITS, 1, &args->opt_t,
1668         NULL);

1670     argc -= i;
1671     argv += i;

1673     if (argc > 1 || (argc == 1 && argv[0].a_type != MDB_TYPE_STRING)) {
1674         mdb_free(args, sizeof (*args));
1675         return (DCMD_USAGE);
1676     }
1677     if (argc == 1)
1678         args->name = argv[0].a_un.a_str;

1680     if ((args->lbolt = mdb_get_lbolt()) == -1) {
1681         mdb_warn("failed to read lbolt");
1682         goto err;
1683     }

1685     if (flags & DCMD_ADDRSPEC) {
1686         if (args->name != NULL) {
1687             mdb_printf("bridge name and address are mutually "
1688                 "exclusive\n");
1689             goto err;
1690         }
1691         if (!args->opt_l && !args->opt_f && !args->opt_t)
1692             mdb_printf("%-?s %-7s %-16s %-7s %-7s\n", "ADDR",
1693                 "PROTECT", "NAME", "NLINKS", "NFWD");
1694         if (do_show_bridge(addr, NULL, args) != WALK_NEXT)
1695             goto err;
1696         mdb_free(args, sizeof (*args));
1697         return (DCMD_OK);
1698     } else {
1699         if ((args->opt_l || args->opt_f || args->opt_t) &&
1700             args->name == NULL) {
1701             mdb_printf("need bridge name or address with -[lft]\n");
1702             goto err;
1703         }
1704         if (mdb_lookup_by_obj("bridge", "inst_list", &sym) == -1) {
1705             mdb_warn("failed to find 'bridge'inst_list");
1706             goto err;
1707         }
1708         if (!args->opt_l && !args->opt_f && !args->opt_t)
1709             mdb_printf("%-?s %-7s %-16s %-7s %-7s %-7s %s\n",
1710                 "ADDR", "PROTECT", "NAME", "NLINKS", "NFWD",
1711                 "NNICKS", "NICK");

```

```
1712         if (mdb_pwalk("list", do_show_bridge, args,
1713             (uintptr_t)sym.st_value) != DCMD_OK)
1714             goto err;
1715         if (!args->found && args->name != NULL) {
1716             mdb_printf("bridge instance %s not found\n",
1717                 args->name);
1718             goto err;
1719         }
1720         mdb_free(args, sizeof (*args));
1721         return (DCMD_OK);
1722     }

1724 err:
1725     mdb_free(args, sizeof (*args));
1726     return (DCMD_ERR);
1727 }

1729 /*
1730  * Support for the "::dladm" dcmd
1731  */
1732 int
1733 dladm(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1734 {
1735     if (argc < 1 || argv[0].a_type != MDB_TYPE_STRING)
1736         return (DCMD_USAGE);

1738     /*
1739      * This could be a bit more elaborate, once we support more of the
1740      * dladm show-* subcommands.
1741      */
1742     argc--;
1743     argv++;
1744     if (strcmp(argv[-1].a_un.a_str, "show-bridge") == 0)
1745         return (dladm_show_bridge(addr, flags, argc, argv));

1747     return (DCMD_USAGE);
1748 }

1750 void
1751 dladm_help(void)
1752 {
1753     mdb_printf("Subcommands:\n"
1754         "  show-bridge [-flt] [<name>]\n"
1755         "  \t Show bridge information; -l for links and -f for "
1756         "forwarding\n"
1757         "  \t entries, and -t for TRILL nicknames. Address is required "
1758         "if name\n"
1759         "  \t is not specified.\n");
1760 }
```


new/usr/src/cmd/mdb/common/modules/genunix/net.h

1

```
*****
2239 Sat Aug 18 10:37:01 2012
new/usr/src/cmd/mdb/common/modules/genunix/net.h
dccb: build fixes, mdb (vfs sonode missing)
*****

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 #ifndef _NET_H
27 #define _NET_H

29 #ifdef __cplusplus
30 extern "C" {
31 #endif

33 extern struct mi_payload_walk_arg_s mi_icmp_arg;
34 extern struct mi_payload_walk_arg_s mi_ill_arg;

36 extern int sonode_walk_init(mdb_walk_state_t *);
37 extern int sonode_walk_step(mdb_walk_state_t *);
38 extern void sonode_walk_fini(mdb_walk_state_t *);
39 extern int mi_walk_init(mdb_walk_state_t *);
40 extern int mi_walk_step(mdb_walk_state_t *);
41 extern void mi_walk_fini(mdb_walk_state_t *);
42 extern int mi_payload_walk_init(mdb_walk_state_t *);
43 extern int mi_payload_walk_step(mdb_walk_state_t *);
44 extern int icmp_stacks_walk_init(mdb_walk_state_t *);
45 extern int icmp_stacks_walk_step(mdb_walk_state_t *);
46 extern int tcp_stacks_walk_init(mdb_walk_state_t *);
47 extern int tcp_stacks_walk_step(mdb_walk_state_t *);
48 extern int udp_stacks_walk_init(mdb_walk_state_t *);
49 extern int udp_stacks_walk_step(mdb_walk_state_t *);
50 extern int dccb_stacks_walk_init(mdb_walk_state_t *);
51 extern int dccb_stacks_walk_step(mdb_walk_state_t *);
52 #endif /* ! codereview */

54 extern int sonode(uintptr_t, uint_t, int, const mdb_arg_t *);
55 extern int mi(uintptr_t, uint_t, int, const mdb_arg_t *);
56 extern int netstat(uintptr_t, uint_t, int, const mdb_arg_t *);
57 extern int dladm(uintptr_t, uint_t, int, const mdb_arg_t *);
58 extern void dladm_help(void);

60 #ifdef __cplusplus
61 }
```

new/usr/src/cmd/mdb/common/modules/genunix/net.h

2

```
62 #endif

64 #endif /* _NET_H */
```

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

1

93952 Sat Aug 18 10:37:02 2012

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

dccp: conn_t

_____unchanged_portion_omitted_____

```
342 /*
343  * Generic network stack walker initialization function. It is used by all
344  * other network stack walkers.
345  */
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;

488     if (wsp->walk_addr == NULL) {
489         if (mdb_lookup_by_obj("ip", "ip_thread_list", &sym) == 0) {
490             wsp->walk_addr = sym.st_value;
491         } else {
492             mdb_warn("unable to locate ip_thread_list\n");
493             return (WALK_ERR);
494         }
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         "\t\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         strcmp(nce_cb->nce_ill_name, ill_name, LIFNAMSIZ) != 0)
775         return (WALK_NEXT);
776
777     if (ncec.ncec_ipversion == IPV6_VERSION) {
778         mdb_printf("%p %5s %-18s %p %6d %N\n",
779             addr, ill_name,
780             nce_l2_addr(ncep, &ill),
781             ncep->nce_fp_mp,
782             ncep->nce_refcnt,
783             &ncep->nce_addr);
784     } else {
785         struct in_addr nceaddr;
786
787         IN6_V4MAPPED_TO_INADDR(&ncep->nce_addr, &nceaddr);
788         mdb_printf("%p %5s %-18s %p %6d %I\n",
789             addr, ill_name,
790             nce_l2_addr(ncep, &ill),
791             ncep->nce_fp_mp,
792             ncep->nce_refcnt,
793             nceaddr.s_addr);
794     }
795
796     return (WALK_NEXT);
797 }
798
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>%s>\n"
955         "%?s %40N\n"
956         "%?s %40d %4d <hb> %s\n",
957         addr, condemned ? "(C)" : "", &irep->ire_setsrc_addr_v6,
958         irep->ire_type, 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>%s>\n"
976         "%?s %40I\n"
977         "%?s %40d %4d <hb> %s\n",
978         addr, condemned ? "(C)" : "", irep->ire_setsrc_addr,
979         irep->ire_type, 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         sctp_hdr_t sctph;

1312         if (mdb_vread(&sctph, sizeof (sctph), addr) == -1) {
1313             mdb_warn("failed to read SCTP header at %p", addr);
1314             return (DCMD_ERR);
1315         }
1316         sctphdr_print(&sctph);
1317         break;
1318     }
1319     default:
1320         break;
1321     }

1323     return (DCMD_OK);
1324 }

1326 static const mdb_bitmask_t ip_flags[] = {
1327     { "DF", IPH_DF, IPH_DF },
1328     { "MF", IPH_MF, IPH_MF },
1329     { NULL, 0, 0 }
1330 };

1332 /* ARGSUSED */
1333 static int
1334 iphdr(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1335 {
1336     uint_t         verbose = FALSE, force = FALSE;
1337     ipha_t         iph[1];
1338     uint16_t       ver, totlen, hdrlen, ipid, off, csum;
1339     uintptr_t      nxt_proto;
1340     char           exp_csum[8];

1342     if (mdb_getopts(argc, argv,

```

```

1343     'v', MDB_OPT_SETBITS, TRUE, &verbose,
1344     'f', MDB_OPT_SETBITS, TRUE, &force, NULL) != argc)
1345         return (DCMD_USAGE);

1347     if (mdb_vread(iph, sizeof (*iph), addr) == -1) {
1348         mdb_warn("failed to read IPv4 header at %p", addr);
1349         return (DCMD_ERR);
1350     }

1352     ver = (iph->ipha_version_and_hdr_length & 0xf0) >> 4;
1353     if (ver != IPV4_VERSION) {
1354         if (ver == IPV6_VERSION) {
1355             return (ip6hdr(addr, flags, argc, argv));
1356         } else if (!force) {
1357             mdb_warn("unknown IP version: %d\n", ver);
1358             return (DCMD_ERR);
1359         }
1360     }

1362     mdb_printf("%<b>IPv4 header%</b>\n");
1363     mdb_printf("%-34s %-34s\n",
1364         "%<u>%-4s %-4s %-5s %-5s %-6s %-5s %-5s %-6s %-8s %-6s%</u>\n",
1365         "SRC", "DST",
1366         "HLEN", "TOS", "LEN", "ID", "OFFSET", "TTL", "PROTO", "CHKSUM",
1367         "EXP-CSUM", "FLGS");

1369     hdrlen = (iph->ipha_version_and_hdr_length & 0x0f) << 2;
1370     mdb_nhconvert(&totlen, &iph->ipha_length, sizeof (totlen));
1371     mdb_nhconvert(&ipid, &iph->ipha_ident, sizeof (ipid));
1372     mdb_nhconvert(&off, &iph->ipha_fragment_offset_and_flags, sizeof (off));
1373     if (hdrlen == IP_SIMPLE_HDR_LENGTH) {
1374         if ((csum = ipcksum(iph, sizeof (*iph))) != 0)
1375             csum = ~(~csum + ~iph->ipha_hdr_checksum);
1376         else
1377             csum = iph->ipha_hdr_checksum;
1378         mdb_snprintf(exp_csum, 8, "%u", csum);
1379     } else {
1380         mdb_snprintf(exp_csum, 8, "<n/a>");
1381     }

1383     mdb_printf("%-34I %-34I\n",
1384         "%-4d %-4d %-5hu %-5hu %-6hu %-5hu %-5hu %-6u %-8s %<5hb>\n",
1385         iph->ipha_src, iph->ipha_dst,
1386         hdrlen, iph->ipha_type_of_service, totlen, ipid,
1387         (off << 3) & 0xffff, iph->ipha_ttl, iph->ipha_protocol,
1388         iph->ipha_hdr_checksum, exp_csum, off, ip_flags);

1390     if (verbose) {
1391         nxt_proto = addr + hdrlen;
1392         return (transport_hdr(iph->ipha_protocol, nxt_proto));
1393     } else {
1394         return (DCMD_OK);
1395     }
1396 }

1398 /* ARGSUSED */
1399 static int
1400 ip6hdr(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1401 {
1402     uint_t         verbose = FALSE, force = FALSE;
1403     ip6_t         iph[1];
1404     int           ver, class, flow;
1405     uint16_t       plen;
1406     uintptr_t      nxt_proto;

1408     if (mdb_getopts(argc, argv,

```

```

1409     'v', MDB_OPT_SETBITS, TRUE, &verbose,
1410     'f', MDB_OPT_SETBITS, TRUE, &force, NULL) != argc)
1411         return (DCMD_USAGE);

1413     if (mdb_vread(iph, sizeof (*iph), addr) == -1) {
1414         mdb_warn("failed to read IPv6 header at %p", addr);
1415         return (DCMD_ERR);
1416     }

1418     ver = (iph->ip6_vfc & 0xf0) >> 4;
1419     if (ver != IPV6_VERSION) {
1420         if (ver == IPV4_VERSION) {
1421             return (iphdr(addr, flags, argc, argv));
1422         } else if (!force) {
1423             mdb_warn("unknown IP version: %d\n", ver);
1424             return (DCMD_ERR);
1425         }
1426     }

1428     mdb_printf("%<b>IPv6 header%</b>\n");
1429     mdb_printf("%<u>%-26s %-26s %4s %7s %5s %3s %3s%</u>\n",
1430         "SRC", "DST", "TCLS", "FLOW-ID", "PLEN", "NXT", "HOP");

1432     class = (iph->ip6_vcf & IPV6_FLOWINFO_TCLASS) >> 20;
1433     mdb_nhconvert(&class, &class, sizeof (class));
1434     flow = iph->ip6_vcf & IPV6_FLOWINFO_FLOWLABEL;
1435     mdb_nhconvert(&flow, &flow, sizeof (flow));
1436     mdb_nhconvert(&plen, &iph->ip6_plen, sizeof (plen));

1438     mdb_printf("%-26N %-26N %4d %7d %5hu %3d %3d\n",
1439         &iph->ip6_src, &iph->ip6_dst,
1440         class, flow, plen, iph->ip6_nxt, iph->ip6_hlim);

1442     if (verbose) {
1443         nxt_proto = addr + sizeof (ip6_t);
1444         return (transport_hdr(iph->ip6_nxt, nxt_proto));
1445     } else {
1446         return (DCMD_OK);
1447     }
1448 }

1450 int
1451 nce(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1452 {
1453     nce_t nce;
1454     nce_cbdata_t nce_cb;
1455     int ipversion = 0;
1456     const char *opt_P = NULL, *opt_ill;

1458     if (mdb_getopts(argc, argv,
1459         'i', MDB_OPT_STR, &opt_ill,
1460         'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1461         return (DCMD_USAGE);

1463     if (opt_P != NULL) {
1464         if (strcmp("v4", opt_P) == 0) {
1465             ipversion = IPV4_VERSION;
1466         } else if (strcmp("v6", opt_P) == 0) {
1467             ipversion = IPV6_VERSION;
1468         } else {
1469             mdb_warn("invalid protocol '%s'\n", opt_P);
1470             return (DCMD_USAGE);
1471         }
1472     }

1474     if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {

```

```

1475         mdb_printf("%<u>%?s %5s %18s %?s %s %s %</u>\n",
1476             "ADDR", "INTF", "LLADDR", "FP_MP", "REFCNT",
1477             "NCE_ADDR");
1478     }

1480     bzero(&nce_cb, sizeof (nce_cb));
1481     if (opt_ill != NULL) {
1482         strcpy(nce_cb.nce_ill_name, opt_ill);
1483     }
1484     nce_cb.nce_ipversion = ipversion;

1486     if (flags & DCMD_ADDRSPEC) {
1487         (void) mdb_vread(&nce, sizeof (nce_t), addr);
1488         (void) nce_format(addr, &nce, &nce_cb);
1489     } else if (mdb_walk("nce", (mdb_walk_cb_t)nce_format, &nce_cb) == -1) {
1490         mdb_warn("failed to walk ire table");
1491         return (DCMD_ERR);
1492     }

1494     return (DCMD_OK);
1495 }

1497 /* ARGSUSED */
1498 static int
1499 dce_format(uintptr_t addr, const dce_t *dcep, void *dce_cb_arg)
1500 {
1501     static const mdb_bitmask_t dmasks[] = {
1502         { "D", DCEF_DEFAULT, DCEF_DEFAULT },
1503         { "P", DCEF_PMTU, DCEF_PMTU },
1504         { "U", DCEF_UINFO, DCEF_UINFO },
1505         { "S", DCEF_TOO_SMALL_PMTU, DCEF_TOO_SMALL_PMTU },
1506         { NULL, 0, 0 }
1507     };
1508     char flagsbuf[2 * A_CNT(dmasks)];
1509     int ipversion = *(int *)dce_cb_arg;
1510     boolean_t condemned = dcep->dce_generation == DCE_GENERATION_CONDEMNED;

1512     if (ipversion != 0 && ipversion != dcep->dce_ipversion)
1513         return (WALK_NEXT);

1515     mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%b", dcep->dce_flags,
1516         dmasks);

1518     switch (dcep->dce_ipversion) {
1519     case IPV4_VERSION:
1520         mdb_printf("%<u>%?p%3s %8s %8d %30I %</u>\n", addr, condemned ?
1521             "(C)" : "", flagsbuf, dcep->dce_pmtu, &dcep->dce_v4addr);
1522         break;
1523     case IPV6_VERSION:
1524         mdb_printf("%<u>%?p%3s %8s %8d %30N %</u>\n", addr, condemned ?
1525             "(C)" : "", flagsbuf, dcep->dce_pmtu, &dcep->dce_v6addr);
1526         break;
1527     default:
1528         mdb_printf("%<u>%?p%3s %8s %8d %30s %</u>\n", addr, condemned ?
1529             "(C)" : "", flagsbuf, dcep->dce_pmtu, "");
1530     }

1532     return (WALK_NEXT);
1533 }

1535 int
1536 dce(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1537 {
1538     dce_t dce;
1539     const char *opt_P = NULL;
1540     const char *zone_name = NULL;

```

```

1541 ip_stack_t *ipst = NULL;
1542 int ipversion = 0;

1544 if (mdb_getopts(argc, argv,
1545 's', MDB_OPT_STR, &zone_name,
1546 'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1547     return (DCMD_USAGE);

1549 /* Follow the specified zone name to find a ip_stack_t*. */
1550 if (zone_name != NULL) {
1551     ipst = zone_to_ips(zone_name);
1552     if (ipst == NULL)
1553         return (DCMD_USAGE);
1554 }

1556 if (opt_P != NULL) {
1557     if (strcmp("v4", opt_P) == 0) {
1558         ipversion = IPV4_VERSION;
1559     } else if (strcmp("v6", opt_P) == 0) {
1560         ipversion = IPV6_VERSION;
1561     } else {
1562         mdb_warn("invalid protocol '%s'\n", opt_P);
1563         return (DCMD_USAGE);
1564     }
1565 }

1567 if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {
1568     mdb_printf("%<u>%?s%3s %8s %8s %30s %</u>\n",
1569 "ADDR", "", "FLAGS", "PMTU", "DST_ADDR");
1570 }

1572 if (flags & DCMD_ADDRSPEC) {
1573     (void) mdb_vread(&dce, sizeof (dce_t), addr);
1574     (void) dce_format(addr, &dce, &ipversion);
1575 } else if (mdb_pwalk("dce", (mdb_walk_cb_t)dce_format, &ipversion,
1576 (uintptr_t)ipst) == -1) {
1577     mdb_warn("failed to walk dce cache");
1578     return (DCMD_ERR);
1579 }

1581 return (DCMD_OK);
1582 }

1584 int
1585 ire(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1586 {
1587     uint_t verbose = FALSE;
1588     ire_t ire;
1589     ire_cbdata_t ire_cb;
1590     int ipversion = 0;
1591     const char *opt_P = NULL;
1592     const char *zone_name = NULL;
1593     ip_stack_t *ipst = NULL;

1595     if (mdb_getopts(argc, argv,
1596 'v', MDB_OPT_SETBITS, TRUE, &verbose,
1597 's', MDB_OPT_STR, &zone_name,
1598 'P', MDB_OPT_STR, &opt_P, NULL) != argc)
1599         return (DCMD_USAGE);

1601     /* Follow the specified zone name to find a ip_stack_t*. */
1602     if (zone_name != NULL) {
1603         ipst = zone_to_ips(zone_name);
1604         if (ipst == NULL)
1605             return (DCMD_USAGE);
1606     }

```

```

1608     if (opt_P != NULL) {
1609         if (strcmp("v4", opt_P) == 0) {
1610             ipversion = IPV4_VERSION;
1611         } else if (strcmp("v6", opt_P) == 0) {
1612             ipversion = IPV6_VERSION;
1613         } else {
1614             mdb_warn("invalid protocol '%s'\n", opt_P);
1615             return (DCMD_USAGE);
1616         }
1617     }

1619     if ((flags & DCMD_LOOPFIRST) || !(flags & DCMD_LOOP)) {
1621         if (verbose) {
1622             mdb_printf("%?s %40s %-20s\n"
1623 "%?s %40s %-20s\n"
1624 "%<u>%?s %40s %4s %-20s %s%</u>\n",
1625 "ADDR", "SRC", "TYPE",
1626 "", "DST", "MARKS",
1627 "", "STACK", "ZONE", "FLAGS", "INTF");
1628         } else {
1629             mdb_printf("%<u>%?s %30s %30s %5s %4s %s%</u>\n",
1630 "ADDR", "SRC", "DST", "STACK", "ZONE", "INTF");
1631         }
1632     }

1634     ire_cb.verbose = (verbose == TRUE);
1635     ire_cb.ire_ipversion = ipversion;

1637     if (flags & DCMD_ADDRSPEC) {
1638         (void) mdb_vread(&ire, sizeof (ire_t), addr);
1639         (void) ire_format(addr, &ire, &ire_cb);
1640     } else if (mdb_pwalk("ire", (mdb_walk_cb_t)ire_format, &ire_cb,
1641 (uintptr_t)ipst) == -1) {
1642         mdb_warn("failed to walk ire table");
1643         return (DCMD_ERR);
1644     }

1646     return (DCMD_OK);
1647 }

1649 static size_t
1650 mi_ospace(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 %8d %?p\n", thw->thw_ipst, mhe->mhe_key,
1811             mhe->mhe_val, th.th_refcnt, th.th_id);
1812     } else if (thw->thw_matchkey == (uintptr_t)mhe->mhe_key) {
1813         int i, j, k;
1814         tr_buf_t *tr;

1816         mdb_printf("Object %p in IP stack %p:\n", mhe->mhe_key,
1817             thw->thw_ipst);
1818         i = th.th_trace_lastref;
1819         mdb_printf("\tThread %p refcnt %d:\n", th.th_id,
1820             th.th_refcnt);
1821         for (j = TR_BUF_MAX; j > 0; j--) {
1822             tr = th.th_trbuf + i;
1823             if (tr->tr_depth == 0 || tr->tr_depth > TR_STACK_DEPTH)
1824                 break;
1825             mdb_printf("\t T%ld:\n", tr->tr_time -
1826                 thw->thw_lbolt);
1827             for (k = 0; k < tr->tr_depth; k++)
1828                 mdb_printf("\t\t%a\n", tr->tr_stack[k]);
1829             if (--i < 0)
1830                 i = TR_BUF_MAX - 1;
1831         }
1832     }
1833     return (WALK_NEXT);
1834 }

1836 /*
1837  * This is called by ::th_trace (via a callback) when walking the th_hash
1838  * list. It calls modent to find the entries.
1839  */
1840 /* ARGSUSED */
1841 static int
1842 th_hash_summary(uintptr_t addr, const void *data, void *private)
1843 {
1844     const th_hash_t *thh = data;
1845     th_walk_data_t *thw = private;

1847     thw->thw_ipst = (uintptr_t)thh->thh_ipst;
1848     return (mdb_pwalk("modent", modent_summary, private,
1849         (uintptr_t)thh->thh_hash));
1850 }

1852 /*
1853  * Print or summarize the th_trace_t structures.
1854  */
1855 static int
1856 th_trace(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1857 {
1858     th_walk_data_t thw;

1860     (void) memset(&thw, 0, sizeof (thw));

1862     if (mdb_getopts(argc, argv,
1863         'n', MDB_OPT_SETBITS, TRUE, &thw.thw_non_zero_only,
1864         NULL) != argc)
1865         return (DCMD_USAGE);

1867     if (!(flags & DCMD_ADDRSPEC)) {
1868         /*
1869          * No address specified. Walk all of the th_hash_t in the
1870          * system, and summarize the th_trace_t entries in each.

```

```

1871 */
1872     mdb_printf("%?s %?s %?s %8s %?s\n",
1873         "IPSTACK", "OBJECT", "TRACE", "REFCNT", "THREAD");
1874     thw.thw_match = B_FALSE;
1875 } else {
1876     thw.thw_match = B_TRUE;
1877     thw.thw_matchkey = addr;
1878
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 }
1890
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 }
1902
1903 static const mdb_dcmd_t dcmds[] = {
1904     { "conn_status", ":",
1905         "display connection structures from ipcl hash tables",
1906         conn_status, conn_status_help },
1907     { "srcid_status", ":",
1908         "display connection structures from ipcl hash tables",
1909         srcid_status },
1910     { "ill", "?[-v] [-P v4 | v6] [-s exclusive-ip-zone-name]",
1911         "display ill_t structures", ill, ill_help },
1912     { "illif", "?[-P v4 | v6]",
1913         "display or filter IP Lower Level InterFace structures", illif,
1914         illif_help },
1915     { "iphdr", ":[-vf]", "display an IPv4 header", iphdr },
1916     { "ip6hdr", ":[-vf]", "display an IPv6 header", ip6hdr },
1917     { "ipif", "?[-v] [-P v4 | v6]", "display ipif structures",
1918         ipif, ipif_help },
1919     { "ire", "?[-v] [-P v4|v6] [-s exclusive-ip-zone-name]",
1920         "display Internet Route Entry structures", ire },
1921     { "nce", "?[-P v4|v6] [-i <interface>]",
1922         "display interface-specific Neighbor Cache structures", nce },
1923     { "ncec", "?[-P v4 | v6]", "display Neighbor Cache Entry structures",
1924         ncec },
1925     { "dce", "?[-P v4|v6] [-s exclusive-ip-zone-name]",
1926         "display Destination Cache Entry structures", dce },
1927     { "squeue", ":[-v]", "print core squeue_t info", squeue,
1928         ip_squeue_help },
1929     { "tcphdr", ":", "display a TCP header", tcphdr },
1930     { "udphdr", ":", "display an UDP header", udphdr },
1931     { "sctphdr", ":", "display an SCTP header", sctphdr },
1932     { "th_trace", "?[-n]", "display th_trace_t structures", th_trace,
1933         th_trace_help },
1934     { NULL }
1935 };

```

```

1937 static const mdb_walker_t walkers[] = {
1938     { "conn_status", "walk list of conn_t structures",
1939       ip_stacks_common_walk_init, conn_status_walk_step, NULL },
1940     { "illif", "walk list of ill interface types for all stacks",
1941       ip_stacks_common_walk_init, illif_walk_step, NULL },
1942     { "illif_stack", "walk list of ill interface types",
1943       illif_stack_walk_init, illif_stack_walk_step,
1944       illif_stack_walk_fini },
1945     { "ill", "walk active ill_t structures for all stacks",
1946       ill_walk_init, ill_walk_step, NULL },
1947     { "ipif", "walk list of ipif structures for all stacks",
1948       ipif_walk_init, ipif_walk_step, NULL },
1949     { "ipif_list", "walk the linked list of ipif structures "
1950       "for a given ill",
1951       ip_list_walk_init, ip_list_walk_step,
1952       ip_list_walk_fini, &ipif_walk_arg },
1953     { "srcid", "walk list of srcid_map structures for all stacks",
1954       ip_stacks_common_walk_init, srcid_walk_step, NULL },
1955     { "srcid_list", "walk list of srcid_map structures for a stack",
1956       ip_list_walk_init, ip_list_walk_step, ip_list_walk_fini,
1957       &srcid_walk_arg },
1958     { "ire", "walk active ire_t structures",
1959       ire_walk_init, ire_walk_step, NULL },
1960     { "ire_next", "walk ire_t structures in the ctable",
1961       ire_next_walk_init, ire_next_walk_step, NULL },
1962     { "nce", "walk active nce_t structures",
1963       nce_walk_init, nce_walk_step, NULL },
1964     { "dce", "walk active dce_t structures",
1965       dce_walk_init, dce_walk_step, NULL },
1966     { "dccp_stacks", "walk all the dccp_stack_t",
1967       ns_walk_init, dccp_stacks_walk_step, NULL },
1968 #endif /* ! codereview */
1969     { "ip_stacks", "walk all the ip_stack_t",
1970       ns_walk_init, ip_stacks_walk_step, NULL },
1971     { "tcp_stacks", "walk all the tcp_stack_t",
1972       ns_walk_init, tcp_stacks_walk_step, NULL },
1973     { "sctp_stacks", "walk all the sctp_stack_t",
1974       ns_walk_init, sctp_stacks_walk_step, NULL },
1975     { "udp_stacks", "walk all the udp_stack_t",
1976       ns_walk_init, udp_stacks_walk_step, NULL },
1977     { "th_hash", "walk all the th_hash_t entries",
1978       th_hash_walk_init, th_hash_walk_step, NULL },
1979     { "ncec", "walk list of ncec structures for all stacks",
1980       ip_stacks_common_walk_init, ncec_walk_step, NULL },
1981     { "ncec_stack", "walk list of ncec structures",
1982       ncec_stack_walk_init, ncec_stack_walk_step,
1983       ncec_stack_walk_fini },
1984     { "udp_hash", "walk list of conn_t structures in ips_ipcl_udp_fanout",
1985       ipcl_hash_walk_init, ipcl_hash_walk_step,
1986       ipcl_hash_walk_fini, &udp_hash_arg },
1987     { "conn_hash", "walk list of conn_t structures in ips_ipcl_conn_fanout",
1988       ipcl_hash_walk_init, ipcl_hash_walk_step,
1989       ipcl_hash_walk_fini, &conn_hash_arg },
1990     { "bind_hash", "walk list of conn_t structures in ips_ipcl_bind_fanout",
1991       ipcl_hash_walk_init, ipcl_hash_walk_step,
1992       ipcl_hash_walk_fini, &bind_hash_arg },
1993     { "proto_hash", "walk list of conn_t structures in "
1994       "ips_ipcl_proto_fanout",
1995       ipcl_hash_walk_init, ipcl_hash_walk_step,
1996       ipcl_hash_walk_fini, &proto_hash_arg },
1997     { "proto_v6_hash", "walk list of conn_t structures in "
1998       "ips_ipcl_proto_fanout_v6",
1999       ipcl_hash_walk_init, ipcl_hash_walk_step,
2000       ipcl_hash_walk_fini, &proto_v6_hash_arg },
2001     { "ilb_stacks", "walk all ilb_stack_t",
2002       ns_walk_init, ilb_stacks_walk_step, NULL },

```

```

2003     { "ilb_rules", "walk ilb rules in a given ilb_stack_t",
2004       ilb_rules_walk_init, ilb_rules_walk_step, NULL },
2005     { "ilb_servers", "walk server in a given ilb_rule_t",
2006       ilb_servers_walk_init, ilb_servers_walk_step, NULL },
2007     { "ilb_nat_src", "walk NAT source table of a given ilb_stack_t",
2008       ilb_nat_src_walk_init, ilb_nat_src_walk_step,
2009       ilb_common_walk_fini },
2010     { "ilb_conns", "walk NAT table of a given ilb_stack_t",
2011       ilb_conn_walk_init, ilb_conn_walk_step, ilb_common_walk_fini },
2012     { "ilb_stickys", "walk sticky table of a given ilb_stack_t",
2013       ilb_sticky_walk_init, ilb_sticky_walk_step,
2014       ilb_common_walk_fini },
2015     { "tcps_sc", "walk all the per CPU stats counters of a tcp_stack_t",
2016       tcps_sc_walk_init, tcps_sc_walk_step, NULL },
2017     { NULL }
2018 };

2020 static const mdb_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->nice_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->nice_dlur_mp) == -1) {
2112             mdb_warn("failed to read nce_dlur_mp at %p",
2113                 nce->nice_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 static int
2198 ncec_format(uintptr_t addr, const ncec_t *ncec, int ipversion)
2199 {
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];

2217 if (mdb_vread(&ill, sizeof (ill), (uintptr_t)ncec->ncec_ill) == -1) {
2218     mdb_warn("failed to read ncec_ill at %p",
2219             ncec->ncec_ill);
2220     return (DCMD_ERR);
2221 }

2223 (void) mdb_readstr(ill_name, MIN(LIFNAMSIZ, ill.ill_name_length),
2224 (uintptr_t)ill.ill_name);

2226 mdb_snprintf(flagsbuf, sizeof (flagsbuf), "%hb",
2227 ncec->ncec_flags, ncec_flags);

2229 if (ipversion != 0 && ncec->ncec_ipversion != ipversion)
2230     return (DCMD_OK);

2232 if (ncec->ncec_ipversion == IPV4_VERSION) {
2233     IN6_V4MAPPED_TO_INADDR(&ncec->ncec_addr, &nceaddr);
2234     mdb_printf("%?p %-20s %-10s "
2235             "%-8s "
2236             "%-5s %I\n",
2237             addr, ncec_l2_addr(ncec, &ill),
2238             ncec_state(ncec->ncec_state),
2239             flagsbuf,
2240             ill_name, nceaddr.s_addr);
2241 } else {
2242     mdb_printf("%?p %-20s %-10s %-8s %-5s %N\n",
2243             addr, ncec_l2_addr(ncec, &ill),
2244             ncec_state(ncec->ncec_state),
2245             flagsbuf,
2246             ill_name, &ncec->ncec_addr);
2247 }

2249 return (DCMD_OK);
2250 }

2252 static uintptr_t
2253 ncec_get_next_hash_tbl(uintptr_t start, int *index, struct ndp_g_s ndp)
2254 {
2255     uintptr_t addr = start;
2256     int i = *index;

2258     while (addr == NULL) {

2260         if (++i >= NCE_TABLE_SIZE)
2261             break;
2262         addr = (uintptr_t)ndp.nce_hash_tbl[i];
2263     }
2264     *index = i;
2265     return (addr);
2266 }

```

```

2268 static int
2269 ncec_walk_step(mdb_walk_state_t *wsp)
2270 {
2271     uintptr_t kaddr4, kaddr6;

2273     kaddr4 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp4);
2274     kaddr6 = wsp->walk_addr + OFFSETOF(ip_stack_t, ips_ndp6);

2276     if (mdb_vread(&kaddr4, sizeof (kaddr4), kaddr4) == -1) {
2277         mdb_warn("can't read ips_ip_cache_table at %p", kaddr4);
2278         return (WALK_ERR);
2279     }
2280     if (mdb_vread(&kaddr6, sizeof (kaddr6), kaddr6) == -1) {
2281         mdb_warn("can't read ips_ip_cache_table at %p", kaddr6);
2282         return (WALK_ERR);
2283     }
2284     if (mdb_pwalk("ncec_stack", wsp->walk_callback, wsp->walk_cbdata,
2285             kaddr4) == -1) {
2286         mdb_warn("couldn't walk 'ncec_stack' for ips_ndp4 %p",
2287             kaddr4);
2288         return (WALK_ERR);
2289     }
2290     if (mdb_pwalk("ncec_stack", wsp->walk_callback,
2291             wsp->walk_cbdata, kaddr6) == -1) {
2292         mdb_warn("couldn't walk 'ncec_stack' for ips_ndp6 %p",
2293             kaddr6);
2294         return (WALK_ERR);
2295     }
2296     return (WALK_NEXT);
2297 }

2299 static uintptr_t
2300 ipcl_hash_get_next_connf_tbl(ipcl_hash_walk_data_t *iw)
2301 {
2302     struct connf_s connf;
2303     uintptr_t addr = NULL, next;
2304     int index = iw->connf_tbl_index;

2306     do {
2307         next = iw->hash_tbl + index * sizeof (struct connf_s);
2308         if (++index >= iw->hash_tbl_size) {
2309             addr = NULL;
2310             break;
2311         }
2312         if (mdb_vread(&connf, sizeof (struct connf_s), next) == -1) {
2313             mdb_warn("failed to read conn_t at %p", next);
2314             return (NULL);
2315         }
2316         addr = (uintptr_t)connf.connf_head;
2317     } while (addr == NULL);
2318     iw->connf_tbl_index = index;
2319     return (addr);
2320 }

2322 static int
2323 ipcl_hash_walk_init(mdb_walk_state_t *wsp)
2324 {
2325     const hash_walk_arg_t *arg = wsp->walk_arg;
2326     ipcl_hash_walk_data_t *iw;
2327     uintptr_t tbladdr;
2328     uintptr_t sizeaddr;

2330     iw = mdb_alloc(sizeof (ipcl_hash_walk_data_t), UM_SLEEP);
2331     iw->conn = mdb_alloc(sizeof (conn_t), UM_SLEEP);
2332     tbladdr = wsp->walk_addr + arg->tbl_off;

```



```

2333     sizeaddr = wsp->walk_addr + arg->size_off;

2335     if (mdb_vread(&iw->hash_tbl, sizeof (uintptr_t), tbladdr) == -1) {
2336         mdb_warn("can't read fanout table addr at %p", tbladdr);
2337         mdb_free(iw->conn, sizeof (conn_t));
2338         mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2339         return (WALK_ERR);
2340     }
2341     if (arg->tbl_off == OFFSETOF(ip_stack_t, ips_ipcl_proto_fanout_v4) ||
2342         arg->tbl_off == OFFSETOF(ip_stack_t, ips_ipcl_proto_fanout_v6)) {
2343         iw->hash_tbl_size = IPPROTO_MAX;
2344     } else {
2345         if (mdb_vread(&iw->hash_tbl_size, sizeof (int),
2346             sizeaddr) == -1) {
2347             mdb_warn("can't read fanout table size addr at %p",
2348                 sizeaddr);
2349             mdb_free(iw->conn, sizeof (conn_t));
2350             mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2351             return (WALK_ERR);
2352         }
2353     }
2354     iw->connf_tbl_index = 0;
2355     wsp->walk_addr = ipcl_hash_get_next_connf_tbl(iw);
2356     wsp->walk_data = iw;

2358     if (wsp->walk_addr != NULL)
2359         return (WALK_NEXT);
2360     else
2361         return (WALK_DONE);
2362 }

2364 static int
2365 ipcl_hash_walk_step(mdb_walk_state_t *wsp)
2366 {
2367     uintptr_t addr = wsp->walk_addr;
2368     ipcl_hash_walk_data_t *iw = wsp->walk_data;
2369     conn_t *conn = iw->conn;
2370     int ret = WALK_DONE;

2372     while (addr != NULL) {
2373         if (mdb_vread(conn, sizeof (conn_t), addr) == -1) {
2374             mdb_warn("failed to read conn_t at %p", addr);
2375             return (WALK_ERR);
2376         }
2377         ret = wsp->walk_callback(addr, iw, wsp->walk_cbdata);
2378         if (ret != WALK_NEXT)
2379             break;
2380         addr = (uintptr_t)conn->conn_next;
2381     }
2382     if (ret == WALK_NEXT) {
2383         wsp->walk_addr = ipcl_hash_get_next_connf_tbl(iw);

2385         if (wsp->walk_addr != NULL)
2386             return (WALK_NEXT);
2387         else
2388             return (WALK_DONE);
2389     }

2391     return (ret);
2392 }

2394 static void
2395 ipcl_hash_walk_fini(mdb_walk_state_t *wsp)
2396 {
2397     ipcl_hash_walk_data_t *iw = wsp->walk_data;

```

```

2399     mdb_free(iw->conn, sizeof (conn_t));
2400     mdb_free(iw, sizeof (ipcl_hash_walk_data_t));
2401 }

2403 /*
2404  * Called with walk_addr being the address of ips_ndp[4,6]
2405  */
2406 static int
2407 ncec_stack_walk_init(mdb_walk_state_t *wsp)
2408 {
2409     ncec_walk_data_t *nw;

2411     if (wsp->walk_addr == NULL) {
2412         mdb_warn("ncec_stack requires ndp_g_s address\n");
2413         return (WALK_ERR);
2414     }

2416     nw = mdb_alloc(sizeof (ncec_walk_data_t), UM_SLEEP);

2418     if (mdb_vread(&nw->ncec_ip_ndp, sizeof (struct ndp_g_s),
2419         wsp->walk_addr) == -1) {
2420         mdb_warn("failed to read 'ip_ndp' at %p",
2421             wsp->walk_addr);
2422         mdb_free(nw, sizeof (ncec_walk_data_t));
2423         return (WALK_ERR);
2424     }

2426     /*
2427      * ncec_get_next_hash_tbl() starts at ++i, so initialize index to -1
2428      */
2429     nw->ncec_hash_tbl_index = -1;
2430     wsp->walk_addr = ncec_get_next_hash_tbl(NULL,
2431         &nw->ncec_hash_tbl_index, nw->ncec_ip_ndp);
2432     wsp->walk_data = nw;

2434     return (WALK_NEXT);
2435 }

2437 static int
2438 ncec_stack_walk_step(mdb_walk_state_t *wsp)
2439 {
2440     uintptr_t addr = wsp->walk_addr;
2441     ncec_walk_data_t *nw = wsp->walk_data;

2443     if (addr == NULL)
2444         return (WALK_DONE);

2446     if (mdb_vread(&nw->ncec, sizeof (ncec_t), addr) == -1) {
2447         mdb_warn("failed to read ncec_t at %p", addr);
2448         return (WALK_ERR);
2449     }

2451     wsp->walk_addr = (uintptr_t)nw->ncec.ncec_next;

2453     wsp->walk_addr = ncec_get_next_hash_tbl(wsp->walk_addr,
2454         &nw->ncec_hash_tbl_index, nw->ncec_ip_ndp);

2456     return (wsp->walk_callback(addr, nw, wsp->walk_cbdata));
2457 }

2459 static void
2460 ncec_stack_walk_fini(mdb_walk_state_t *wsp)
2461 {
2462     mdb_free(wsp->walk_data, sizeof (ncec_walk_data_t));
2463 }

```

```

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%s+--> %3d %-18s "
2635                  "references from active threads\n",
2636                  sbuf, sbuf, ill->ill_refcnt, "ill_refcnt");
2637         mdb_printf("%*s %7d %-18s ires referencing this ill\n",
2638                  strlen(sbuf), "", ill->ill_ire_cnt, "ill_ire_cnt");
2639         mdb_printf("%*s %7d %-18s nces referencing this ill\n",
2640                  strlen(sbuf), "", ill->ill_nce_cnt, "ill_nce_cnt");
2641         mdb_printf("%*s %7d %-18s ncecs referencing this ill\n",
2642                  strlen(sbuf), "", ill->ill_ncec_cnt, "ill_ncec_cnt");
2643         mdb_printf("%*s %7d %-18s ilms referencing this ill\n",
2644                  strlen(sbuf), "", ill->ill_ilm_cnt, "ill_ilm_cnt");
2645     } else {
2646         mdb_printf("%4d %-?p %-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-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("%-7s %-10s %-3s %-7s %-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("%-7s %-10s %6s %-7s %-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(fmask)];
2888 char bitfields[A_CNT(fmask)];
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:\n"
3019         "\n\tUNN -> UNNUMBERED, DHCP -> DHCPRUNNING, PRIV -> PRIVATE, "
3020         "\n\tNOXMT -> NOXMIT, NOCL -> NOLOCAL, DEPR -> DEPRECATED, "
3021         "\n\tPREF -> PREFERRED, TEMP -> TEMPORARY, ACONF -> ADDRCONF, "
3022         "\n\tANY -> ANYCAST, NFAIL -> NOFAILOVER, "
3023         "\n\tADR -> ipif_addr_ready, MU -> ipif_multicast_up, "
3024         "\n\tWU -> ipif_was_up, WD -> ipif_was_dup, "
3025         "JA -> ipif_joined_allhosts.\n\n");
3026     mdb_printf("Options:\n");
3027     mdb_printf("\t-t-P v4 | v6\n");
3028     "\tfilter ipif structures on ills for the specified protocol\n");
3029 }

3031 static int
3032 conn_status_walk_fanout(uintptr_t addr, mdb_walk_state_t *wsp,
3033     const char *walkname)
3034 {
3035     if (mdb_pwalk(walkname, wsp->walk_callback, wsp->walk_cbdata,
3036         addr) == -1) {
3037         mdb_warn("couldn't walk '%s' at %p", walkname, addr);
3038         return (WALK_ERR);
3039     }
3040     return (WALK_NEXT);
3041 }

3043 static int
3044 conn_status_walk_step(mdb_walk_state_t *wsp)
3045 {
3046     uintptr_t addr = wsp->walk_addr;

3048     (void) conn_status_walk_fanout(addr, wsp, "udp_hash");
3049     (void) conn_status_walk_fanout(addr, wsp, "conn_hash");
3050     (void) conn_status_walk_fanout(addr, wsp, "bind_hash");
3051     (void) conn_status_walk_fanout(addr, wsp, "proto_hash");
3052     (void) conn_status_walk_fanout(addr, wsp, "proto_v6_hash");
3053     return (WALK_NEXT);
3054 }

3056 /* ARGSUSED */
3057 static int
3058 conn_status_cb(uintptr_t addr, const void *walk_data,

```

```

3059     void *private)
3060 {
3061     netstack_t nss;
3062     char src_addrstr[INET6_ADDRSTRLEN];
3063     char rem_addrstr[INET6_ADDRSTRLEN];
3064     const ipcl_hash_walk_data_t *iw = walk_data;
3065     conn_t *conn = iw->conn;

3067     if (mdb_vread(conn, sizeof (conn_t), addr) == -1) {
3068         mdb_warn("failed to read conn_t at %p", addr);
3069         return (WALK_ERR);
3070     }
3071     if (mdb_vread(&nss, sizeof (nss),
3072         (uintptr_t)conn->conn_netstack) == -1) {
3073         mdb_warn("failed to read netstack_t %p",
3074             conn->conn_netstack);
3075         return (WALK_ERR);
3076     }
3077     mdb_printf("%-?p %-?p %?d %?d\n", addr, conn->conn_wq,
3078         nss.netstack_stackid, conn->conn_zoneid);

3080     if (conn->conn_family == AF_INET6) {
3081         mdb_snprintf(src_addrstr, sizeof (rem_addrstr), "%N",
3082             &conn->conn_laddr_v6);
3083         mdb_snprintf(rem_addrstr, sizeof (rem_addrstr), "%N",
3084             &conn->conn_faddr_v6);
3085     } else {
3086         mdb_snprintf(src_addrstr, sizeof (src_addrstr), "%I",
3087             V4_PART_OF_V6((conn->conn_laddr_v6)));
3088         mdb_snprintf(rem_addrstr, sizeof (rem_addrstr), "%I",
3089             V4_PART_OF_V6((conn->conn_faddr_v6)));
3090     }
3091     mdb_printf("%s:%-5d\n%s:%-5d\n",
3092         src_addrstr, conn->conn_lport, rem_addrstr, conn->conn_fport);
3093     return (WALK_NEXT);
3094 }

3096 static void
3097 conn_header(void)
3098 {
3099     mdb_printf("%-?s %-?s %?s %?s\n%s\n%s\n",
3100         "ADDR", "WQ", "STACK", "ZONE", "SRC:PORT", "DEST:PORT");
3101     mdb_printf("%<u>%80s</u>\n", "");
3102 }

3104 /*ARGSUSED*/
3105 static int
3106 conn_status(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
3107 {
3108     conn_header();
3109     if (flags & DCMD_ADDRSPEC) {
3110         (void) conn_status_cb(addr, NULL, NULL);
3111     } else {
3112         if (mdb_walk("conn_status", (mdb_walk_cb_t)conn_status_cb,
3113             NULL) == -1) {
3114             mdb_warn("failed to walk conn_fanout");
3115             return (DCMD_ERR);
3116         }
3117     }
3118     return (DCMD_OK);
3119 }

3121 static void
3122 conn_status_help(void)
3123 {
3124     mdb_printf("Prints conn_t structures from the following hash tables: "

```

```

3125     "\n\tips_ipcl_udp_fanout\n\tips_ipcl_bind_fanout"
3126     "\n\tips_ipcl_conn_fanout\n\tips_ipcl_proto_fanout_v4"
3127     "\n\tips_ipcl_proto_fanout_v6\n");
3128 }

3130 static int
3131 srcid_walk_step(mdb_walk_state_t *wsp)
3132 {
3133     if (mdb_pwalk("srcid_list", wsp->walk_callback, wsp->walk_cbdata,
3134         wsp->walk_addr) == -1) {
3135         mdb_warn("can't walk 'srcid_list'");
3136         return (WALK_ERR);
3137     }
3138     return (WALK_NEXT);
3139 }

3141 /* ARGSUSED */
3142 static int
3143 srcid_status_cb(uintptr_t addr, const void *walk_data,
3144     void *private)
3145 {
3146     srcid_map_t smp;

3148     if (mdb_vread(&smp, sizeof (srcid_map_t), addr) == -1) {
3149         mdb_warn("failed to read srcid_map at %p", addr);
3150         return (WALK_ERR);
3151     }
3152     mdb_printf("%-?p %3d %4d %6d %N\n",
3153         addr, smp.sm_srcid, smp.sm_zoneid, smp.sm_refcnt,
3154         &smp.sm_addr);
3155     return (WALK_NEXT);
3156 }

3158 static void
3159 srcid_header(void)
3160 {
3161     mdb_printf("%-?s %3s %4s %6s %s\n",
3162         "ADDR", "ID", "ZONE", "REFCNT", "IPADDR");
3163     mdb_printf("%<u>%80s%</u>\n", "");
3164 }

3166 /*ARGSUSED*/
3167 static int
3168 srcid_status(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
3169 {
3170     srcid_header();
3171     if (flags & DCMD_ADDRSPEC) {
3172         (void) srcid_status_cb(addr, NULL, NULL);
3173     } else {
3174         if (mdb_walk("srcid", (mdb_walk_cb_t)srcid_status_cb,
3175             NULL) == -1) {
3176             mdb_warn("failed to walk srcid_map");
3177             return (DCMD_ERR);
3178         }
3179     }
3180     return (DCMD_OK);
3181 }

3183 static int
3184 ilb_stacks_walk_step(mdb_walk_state_t *wsp)
3185 {
3186     return (ns_walk_step(wsp, NS_ILB));
3187 }

3189 static int
3190 ilb_rules_walk_init(mdb_walk_state_t *wsp)

```

```

3191 {
3192     ilb_stack_t ilbs;

3194     if (wsp->walk_addr == NULL)
3195         return (WALK_ERR);

3197     if (mdb_vread(&ilbs, sizeof (ilbs), wsp->walk_addr) == -1) {
3198         mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3199         return (WALK_ERR);
3200     }
3201     if ((wsp->walk_addr = (uintptr_t)ilbs.ilbs_rule_head) != NULL)
3202         return (WALK_NEXT);
3203     else
3204         return (WALK_DONE);
3205 }

3207 static int
3208 ilb_rules_walk_step(mdb_walk_state_t *wsp)
3209 {
3210     ilb_rule_t rule;
3211     int status;

3213     if (mdb_vread(&rule, sizeof (rule), wsp->walk_addr) == -1) {
3214         mdb_warn("failed to read ilb_rule_t at %p", wsp->walk_addr);
3215         return (WALK_ERR);
3216     }
3217     status = wsp->walk_callback(wsp->walk_addr, &rule, wsp->walk_cbdata);
3218     if (status != WALK_NEXT)
3219         return (status);
3220     if ((wsp->walk_addr = (uintptr_t)rule.ir_next) == NULL)
3221         return (WALK_DONE);
3222     else
3223         return (WALK_NEXT);
3224 }

3226 static int
3227 ilb_servers_walk_init(mdb_walk_state_t *wsp)
3228 {
3229     ilb_rule_t rule;

3231     if (wsp->walk_addr == NULL)
3232         return (WALK_ERR);

3234     if (mdb_vread(&rule, sizeof (rule), wsp->walk_addr) == -1) {
3235         mdb_warn("failed to read ilb_rule_t at %p", wsp->walk_addr);
3236         return (WALK_ERR);
3237     }
3238     if ((wsp->walk_addr = (uintptr_t)rule.ir_servers) != NULL)
3239         return (WALK_NEXT);
3240     else
3241         return (WALK_DONE);
3242 }

3244 static int
3245 ilb_servers_walk_step(mdb_walk_state_t *wsp)
3246 {
3247     ilb_server_t server;
3248     int status;

3250     if (mdb_vread(&server, sizeof (server), wsp->walk_addr) == -1) {
3251         mdb_warn("failed to read ilb_server_t at %p", wsp->walk_addr);
3252         return (WALK_ERR);
3253     }
3254     status = wsp->walk_callback(wsp->walk_addr, &server, wsp->walk_cbdata);
3255     if (status != WALK_NEXT)
3256         return (status);

```

```

3257     if ((wsp->walk_addr = (uintptr_t)server.isr_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;

3523     if (wsp->walk_addr == NULL)
3524         return (WALK_ERR);

3526     sticky_walk = mdb_alloc(sizeof (ilb_walk_t), UM_SLEEP);
3527     if (mdb_vread(&sticky_walk->ilbs, sizeof (sticky_walk->ilbs),
3528         wsp->walk_addr) == -1) {
3529         mdb_warn("failed to read ilb_stack_t at %p", wsp->walk_addr);
3530         mdb_free(sticky_walk, sizeof (ilb_walk_t));
3531         return (WALK_ERR);
3532     }

3534     if (sticky_walk->ilbs.ilbs_sticky_hash == NULL) {
3535         mdb_free(sticky_walk, sizeof (ilb_walk_t));
3536         return (WALK_DONE);
3537     }

3539     wsp->walk_data = sticky_walk;
3540     for (i = 0; i < sticky_walk->ilbs.ilbs_sticky_hash_size; i++) {
3541         list_t head;
3542         char *khead;

3544         /* Read in the nsh_head in the i-th element of the array. */
3545         khead = (char *)sticky_walk->ilbs.ilbs_sticky_hash + i *
3546             sizeof (ilb_sticky_hash_t);
3547         if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3548             mdb_warn("failed to read ilbs_sticky_hash at %p\n",
3549                 khead);
3550             return (WALK_ERR);
3551         }

3553         /*
3554          * Note that list_next points to a kernel address and we need
3555          * to compare list_next with the kernel address of the list
3556          * head. So we need to calculate the address manually.
3557          */
3558         if ((char *)head.list_head.list_next != khead +
3559             offsetof(list_t, list_head)) {
3560             st = list_object(&head, head.list_head.list_next);
3561             break;
3562         }
3563     }

3565     if (st == NULL)
3566         return (WALK_DONE);

3568     wsp->walk_addr = (uintptr_t)st;
3569     sticky_walk->idx = i;
3570     return (WALK_NEXT);
3571 }

3573 static int
3574 ilb_sticky_walk_step(mdb_walk_state_t *wsp)
3575 {
3576     int status;
3577     ilb_sticky_t st, *st_next;
3578     ilb_walk_t *sticky_walk;
3579     ilb_stack_t *ilbs;
3580     list_t head;
3581     char *khead;
3582     int i;

3584     if (mdb_vread(&st, sizeof (ilb_sticky_t), wsp->walk_addr) == -1) {
3585         mdb_warn("failed to read ilb_sticky_t at %p", wsp->walk_addr);
3586         return (WALK_ERR);

```

```

3587     }

3589     status = wsp->walk_callback(wsp->walk_addr, &st, wsp->walk_cbdata);
3590     if (status != WALK_NEXT)
3591         return (status);

3593     sticky_walk = (ilb_walk_t *)wsp->walk_data;
3594     ilbs = &sticky_walk->ilbs;
3595     i = sticky_walk->idx;

3597     /* Read in the nsh_head in the i-th element of the array. */
3598     khead = (char *)ilbs->ilbs_sticky_hash + i * sizeof (ilb_sticky_hash_t);
3599     if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3600         mdb_warn("failed to read ilbs_sticky_hash at %p\n", khead);
3601         return (WALK_ERR);
3602     }

3604     /*
3605      * Check if there is still entry in the current list.
3606      *
3607      * Note that list_next points to a kernel address and we need to
3608      * compare list_next with the kernel address of the list head.
3609      * So we need to calculate the address manually.
3610      */
3611     if ((char *)st.list.list_next != khead + offsetof(list_t,
3612         list_head)) {
3613         wsp->walk_addr = (uintptr_t)list_object(&head,
3614             st.list.list_next);
3615         return (WALK_NEXT);
3616     }

3618     /* Start with the next bucket in the array. */
3619     st_next = NULL;
3620     for (i++; i < ilbs->ilbs_nat_src_hash_size; i++) {
3621         khead = (char *)ilbs->ilbs_sticky_hash + i *
3622             sizeof (ilb_sticky_hash_t);
3623         if (mdb_vread(&head, sizeof (list_t), (uintptr_t)khead) == -1) {
3624             mdb_warn("failed to read ilbs_sticky_hash at %p\n",
3625                 khead);
3626             return (WALK_ERR);
3627         }

3629         if ((char *)head.list_head.list_next != khead +
3630             offsetof(list_t, list_head)) {
3631             st_next = list_object(&head,
3632                 head.list_head.list_next);
3633             break;
3634         }
3635     }

3637     if (st_next == NULL)
3638         return (WALK_DONE);

3640     wsp->walk_addr = (uintptr_t)st_next;
3641     sticky_walk->idx = i;
3642     return (WALK_NEXT);
3643 }

```

new/usr/src/lib/libdtrace/Makefile.com

1

5931 Sat Aug 18 10:37:02 2012
new/usr/src/lib/libdtrace/Makefile.com
dccb: basic dtrace

```
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 # Copyright (c) 2011 by Delphix. All rights reserved.
24 #

26 LIBRARY = libdtrace.a
27 VERS = .1

29 LIBSRCS = \
30     dt_aggregate.c \
31     dt_as.c \
32     dt_buf.c \
33     dt_cc.c \
34     dt_cg.c \
35     dt_consume.c \
36     dt_decl.c \
37     dt_dis.c \
38     dt_dof.c \
39     dt_error.c \
40     dt_errtags.c \
41     dt_handle.c \
42     dt_ident.c \
43     dt_inttab.c \
44     dt_link.c \
45     dt_list.c \
46     dt_open.c \
47     dt_options.c \
48     dt_program.c \
49     dt_map.c \
50     dt_module.c \
51     dt_names.c \
52     dt_parser.c \
53     dt_pcb.c \
54     dt_pid.c \
55     dt_pragma.c \
56     dt_print.c \
57     dt_printf.c \
58     dt_proc.c \
59     dt_provider.c \
60     dt_regset.c \
61     dt_string.c \
```

new/usr/src/lib/libdtrace/Makefile.com

2

```
62     dt_strtab.c \
63     dt_subr.c \
64     dt_work.c \
65     dt_xlator.c
```

```
67 LIBISARCS = \
68     dt_isadep.c
```

```
70 OBJECTS = dt_lex.o dt_grammar.o $(MACHOBJS) $(LIBSRCS:%.c=%.o) $(LIBISARCS:%.c=
```

```
72 DRTISRC = drti.c
73 DRTIOBJ = $(DRTISRC:%.c=%.o)
```

```
75 DLIBSRCS += \
76     dccb.d \
77 #endif /* ! codereview */
78     errno.d \
79     fc.d \
80     io.d \
81     ip.d \
82     iscsit.d \
83     net.d \
84     nfs.d \
85     nfssrv.d \
86     procfs.d \
87     regs.d \
88     sched.d \
89     signal.d \
90     scsi.d \
91     srp.d \
92     sysevent.d \
93     tcp.d \
94     udp.d \
95     unistd.d
```

```
97 include .././Makefile.lib
```

```
99 SRCS = $(LIBSRCS:%.c=../common/%.c) $(LIBISARCS:%.c=../$(MACH)/%.c)
100 LIBS = $(DYNLIB) $(LINTLIB)
```

```
102 SRCDIR = ../common
```

```
104 CLEANFILES += dt_lex.c dt_grammar.c dt_grammar.h y.output
105 CLEANFILES += ../common/procfs.sed ../common/procfs.d
106 CLEANFILES += ../common/io.sed ../common/io.d
107 CLEANFILES += ../common/ip.sed ../common/ip.d
108 CLEANFILES += ../common/net.sed ../common/net.d
109 CLEANFILES += ../common/errno.d ../common/signal.d
110 CLEANFILES += ../common/dt_errtags.c ../common/dt_names.c
111 CLEANFILES += ../common/sysevent.sed ../common/sysevent.d
112 CLEANFILES += ../common/tcp.sed ../common/tcp.d
113 CLEANFILES += ../common/udp.sed ../common/udp.d
114 CLEANFILES += ../common/dccb.sed ../common/dccb.d
115 #endif /* ! codereview */
```

```
117 CLOBBERFILES = drti.o
```

```
119 CPPFLAGS += -I../common -I.
120 CFLAGS += $(CCVERBOSE) $(C_BIGPICFLAGS)
121 CFLAGS64 += $(CCVERBOSE) $(C_BIGPICFLAGS)
122 YYCFLAGS =
123 LDLIBS += -lgen -lproc -lrtl_d_db -lnsl -lsocket -lctf -lelf -lc
124 DRTILDLIBS = $(LDLIBS.lib) -lc
```

```
126 yydebug := YYCFLAGS += -DYYDEBUG
```

```

128 $(LINTLIB) := SRCS = $(SRCDIR)/$(LINTSRC)
130 LFLAGS = -t -v
131 YFLAGS = -d -v
133 ROOTDLIBDIR = $(ROOT)/usr/lib/dtrace
134 ROOTDLIBDIR64 = $(ROOT)/usr/lib/dtrace/64
136 ROOTDLIBS = $(DLIBSRCS:%=$(ROOTDLIBDIR)/%)
137 ROOTDOBS = $(ROOTDLIBDIR)/$(DRTIOBJ)
138 ROOTDOBS64 = $(ROOTDLIBDIR64)/$(DRTIOBJ)
140 .KEEP_STATE:
142 all: $(LIBS) $(DRTIOBJ)
144 lint: lintdrti lintcheck
146 lintdrti: ../common/$(DRTISRC)
147          $(LINT.c) ../common/$(DRTISRC) $(DRTILDLIBS)
149 dt_lex.c: $(SRCDIR)/dt_lex.l dt_grammar.h
150          $(LEX) $(LFLAGS) $(SRCDIR)/dt_lex.l > $@
152 dt_grammar.c dt_grammar.h: $(SRCDIR)/dt_grammar.y
153          $(YACC) $(YFLAGS) $(SRCDIR)/dt_grammar.y
154          @mv y.tab.h dt_grammar.h
155          @mv y.tab.c dt_grammar.c
157 pics/dt_lex.o pics/dt_grammar.o := CFLAGS += $(YYCFLAGS)
158 pics/dt_lex.o pics/dt_grammar.o := CFLAGS64 += $(YYCFLAGS)
160 pics/dt_lex.o pics/dt_grammar.o := CERRWARN += -erroff=E_STATEMENT_NOT_REACHED
161 pics/dt_lex.o pics/dt_grammar.o := CCVERBOSE =
163 ../common/dt_errtags.c: ../common/mkerrtags.sh ../common/dt_errtags.h
164          sh ../common/mkerrtags.sh < ../common/dt_errtags.h > $@
166 ../common/dt_names.c: ../common/mknames.sh $(SRC)/uts/common/sys/dtrace.h
167          sh ../common/mknames.sh < $(SRC)/uts/common/sys/dtrace.h > $@
169 ../common/errno.d: ../common/mkerrno.sh $(SRC)/uts/common/sys/errno.h
170          sh ../common/mkerrno.sh < $(SRC)/uts/common/sys/errno.h > $@
172 ../common/signal.d: ../common/mksignal.sh $(SRC)/uts/common/sys/iso/signal_iso.h
173          sh ../common/mksignal.sh < $(SRC)/uts/common/sys/iso/signal_iso.h > $@
175 ../common/%.sed: ../common/%.sed.in
176          $(COMPILE.cpp) -D_KERNEL $< | tr -d ' ' | tr '"" '@' | \
177          sed 's/\&\\&/g' | grep '^s/' > $@
179 ../common/procfs.d: ../common/procfs.sed ../common/procfs.d.in
180          sed -f ../common/procfs.sed < ../common/procfs.d.in > $@
182 ../common/io.d: ../common/io.sed ../common/io.d.in
183          sed -f ../common/io.sed < ../common/io.d.in > $@
185 ../common/ip.d: ../common/ip.sed ../common/ip.d.in
186          sed -f ../common/ip.sed < ../common/ip.d.in > $@
188 ../common/net.d: ../common/net.sed ../common/net.d.in
189          sed -f ../common/net.sed < ../common/net.d.in > $@
191 ../common/sysevent.d: ../common/sysevent.sed ../common/sysevent.d.in
192          sed -f ../common/sysevent.sed < ../common/sysevent.d.in > $@

```

```

194 ../common/tcp.d: ../common/tcp.sed ../common/tcp.d.in
195          sed -f ../common/tcp.sed < ../common/tcp.d.in > $@
197 ../common/udp.d: ../common/udp.sed ../common/udp.d.in
198          sed -f ../common/udp.sed < ../common/udp.d.in > $@
200 ../common/dccp.d: ../common/dccp.sed ../common/dccp.d.in
201          sed -f ../common/dccp.sed < ../common/dccp.d.in > $@
203 #endif /* ! codereview */
204 pics/%.o: ../$(MACH)/%.c
205          $(COMPILE.c) -o $@ $<
206          $(POST_PROCESS_O)
208 pics/%.o: ../$(MACH)/%.s
209          $(COMPILE.s) -o $@ $<
210          $(POST_PROCESS_O)
212 %.o: ../common/%.c
213          $(COMPILE.c) -o $@ $<
214          $(POST_PROCESS_O)
216 $(ROOTDLIBDIR):
217          $(INS.dir)
219 $(ROOTDLIBDIR64): $(ROOTDLIBDIR)
220          $(INS.dir)
222 $(ROOTDLIBDIR)/%.d: ../common/%.d
223          $(INS.file)
225 $(ROOTDLIBDIR)/%.d: ../$(MACH)/%.d
226          $(INS.file)
228 $(ROOTDLIBDIR)/%.d: %.d
229          $(INS.file)
231 $(ROOTDLIBDIR)/%.o: %.o
232          $(INS.file)
234 $(ROOTDLIBDIR64)/%.o: %.o
235          $(INS.file)
237 $(ROOTDLIBS): $(ROOTDLIBDIR)
239 $(ROOTDOBS): $(ROOTDLIBDIR)
241 $(ROOTDOBS64): $(ROOTDLIBDIR64)
243 include ../../Makefile.targ

```

new/usr/src/lib/libdtrace/common/dccp.d.in

1

```
*****
6829 Sat Aug 18 10:37:03 2012
new/usr/src/lib/libdtrace/common/dccp.d.in
dccp: basic dtrace
*****

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 #pragma D depends_on module unix
26 #pragma D depends_on provider dccp

28 inline int32_t DCCP_STATE_CLOSED = @DCCPS_CLOSED@;
29 #pragma D binding "1.10" DCCP_STATE_CLOSED
30 inline int32_t DCCP_STATE_BOUND = @DCCPS_BOUND@;
31 #pragma D binding "1.10" DCCP_STATE_BOUND
32 inline int32_t DCCP_STATE_REQUEST = @DCCPS_REQUEST@;
33 #pragma D binding "1.10" DCCP_STATE_REQUEST
34 inline int32_t DCCP_STATE_LISTEN = @DCCPS_LISTEN@;
35 #pragma D binding "1.10" DCCP_STATE_LISTEN
36 inline int32_t DCCP_STATE_PARTOPEN = @DCCPS_PARTOPEN@;
37 #pragma D binding "1.10" DCCP_STATE_PARTOPEN
38 inline int32_t DCCP_STATE_RESPOND = @DCCPS_RESPOND@;
39 #pragma D binding "1.10" DCCP_STATE_RESPOND
40 inline int32_t DCCP_STATE_OPEN = @DCCPS_OPEN@;
41 #pragma D binding "1.10" DCCP_STATE_OPEN
42 inline int32_t DCCP_STATE_CLOSING = @DCCPS_CLOSING@;
43 #pragma D binding "1.10" DCCP_STATE_CLOSING
44 inline int32_t DCCP_STATE_CLOSEREQ = @DCCPS_CLOSEREQ@;
45 #pragma D binding "1.10" DCCP_STATE_CLOSEREQ
46 inline int32_t DCCP_STATE_TIMEWAIT = @DCCPS_TIMEWAIT@;
47 #pragma D binding "1.10" DCCP_STATE_TIMEWAIT

49 /*
50  * Convert a DCCP state value to a string.
51  */
52 inline string dccp_state_string[int32_t state] =
53     state == DCCP_STATE_CLOSED ? "state-closed" :
54     state == DCCP_STATE_BOUND ? "state-bound" :
55     state == DCCP_STATE_REQUEST ? "state-request" :
56     state == DCCP_STATE_LISTEN ? "state-listen" :
57     state == DCCP_STATE_PARTOPEN ? "state-partopen" :
58     state == DCCP_STATE_RESPOND ? "state-respond" :
59     state == DCCP_STATE_OPEN ? "state-open" :
60     state == DCCP_STATE_CLOSING ? "state-closing" :
61     state == DCCP_STATE_CLOSEREQ ? "state-closereq" :
```

new/usr/src/lib/libdtrace/common/dccp.d.in

2

```
62     state == DCCP_STATE_TIMEWAIT ? "state-timewait" :
63     "<unknown>";
64 #pragma D binding "1.10" dccp_state_string

66 /*
67  * dccpinfo is the DCCP header fields.
68  */
69 typedef struct dccpinfo {
70     uint16_t dccp_sport;           /* source port */
71     uint16_t dccp_dport;           /* destination port */
72     uint32_t dccp_seq;             /* sequence number */
73     uint8_t dccp_offset;           /* data offset, in bytes */
74     uint16_t dccp_checksum;        /* checksum */
75     dccph_t *dccp_hdr;             /* raw DCCP header */
76 } dccpinfo_t;

78 /*
79  * dccpsinfo contains stable DCCP details from dccp_t.
80  */
81 typedef struct dccpsinfo {
82     uintptr_t dccps_addr;
83     int dccps_local;               /* Is delivered locally, boolean */
84     int dccps_active;             /* Active open (from here), boolean */
85     uint16_t dccps_lport;          /* Local port */
86     uint16_t dccps_rport;          /* Remote port */
87     string dccps_laddr;            /* Local address, as a string */
88     string dccps_raddr;            /* Remote address, as a string */
89     int32_t dccps_state;           /* DCCP state */
90     uint32_t dccps_iss;            /* Initial sequence # sent */
91 } dccpsinfo_t;

93 /*
94  * dccplsinfo provides the old dccp state for state changes.
95  */
96 typedef struct dccplsinfo {
97     int32_t dccps_state;           /* Previous DCCP state */
98 } dccplsinfo_t;

100 /*
101  * __dtrace_tcp_tcpht is used by the tcpinfo_t * translator to take either
102  * the non-NULL tcpht * passed in or, if it is NULL, uses arg3 (tcp_t *)
103  * from the tcp::send and tcp::receive probes and translates the tcp_t *
104  * into the tcpinfo_t. When no headers are available - as is the case for
105  * TCP fusion tcp::send and tcp::receive - this allows us to present the
106  * consumer with header data based on tcp_t * content and hide TCP fusion
107  * implementation details.
108  */
109 typedef dccph_t * __dtrace_dccp_dccph_t;

111 #pragma D binding "1.10" translator
112 translator dccpinfo_t < dccph_t *T > {
113     dccp_sport = ntohs(*(uint16_t *)T->dh_lport);
114     dccp_dport = ntohs(*(uint16_t *)T->dh_fport);
115     dccp_seq = ntohl(*(uint32_t *)T->dh_seq); /* XXX */
116     dccp_offset = (*(uint8_t *)T->dh_offset & 0xf0) >> 2;
117     dccp_checksum = ntohs(*(uint16_t *)T->dh_sum);
118     dccp_hdr = T;
119 };

121 #pragma D binding "1.10" translator
122 translator dccpinfo_t < __dtrace_dccp_dccph_t *T > {
123     dccp_sport =
124         T != NULL ? ntohs(*(uint16_t *)((dccph_t *)T)->dh_lport) :
125         arg3 != NULL && probename == "send" ?
126         ntohs(((dccp_t *)arg3)->dccp_connnp->u_port.conn_u_ports.conn_u_lport)
127         : arg3 != NULL && probename == "receive" ?
```

```

128     ntohs(((dccp_t *)arg3)->dccp_connp->u_port.connu_ports.connu_fport)
129     0;
130     dccp_dport =
131     T != NULL ? ntohs(*(uint16_t *)((dccph_t *)T)->dh_fport) :
132     arg3 != NULL && probename == "send" ?
133     ntohs(((dccp_t *)arg3)->dccp_connp->u_port.connu_ports.connu_fport)
134     arg3 != NULL && probename == "receive" ?
135     ntohs(((dccp_t *)arg3)->dccp_connp->u_port.connu_ports.connu_lport)
136     0;
137     dccp_seq =
138     T != NULL ? ntohl(*(uint32_t *)((dccph_t *)T)->dh_seq) :
139     0;
140     dccp_offset = T != NULL ?
141     (*(uint8_t *)((dccph_t *)T)->dh_offset & 0xf0) >> 2 :
142     @DCCP_MIN_HEADER_LENGTH@;
143     dccp_checksum = T != NULL ? ntohs(*(uint16_t *)((dccph_t *)T)->dh_sum) :
144     0;
145     dccp_hdr = NULL;
146 };

148 #pragma D binding "1.10" translator
149 translator dccpsinfo_t < dccp_t *T > {
150     dccps_addr = (uintptr_t)T;
151     /*
152     * The following two members should just use tcp_t->tcp_loopback
153     * and tcp_t->tcp_active_open, however these are bit fields and
154     * can't be used until CR 6876830 is fixed. Meanwhile we source
155     * them a different way.
156     */
157     dccps_local = T ? T->dccp_ipha ?
158     T->dccp_ipha->ipha_src == T->dccp_ipha->ipha_dst : 1 : 0;
159     dccps_active = T ? !T->dccp_saved_listener : 0;
160     dccps_lport = T ?
161     ntohs(T->dccp_connp->u_port.connu_ports.connu_lport) : 0;
162     dccps_rport = T ?
163     ntohs(T->dccp_connp->u_port.connu_ports.connu_fport) : 0;
164     dccps_laddr = T ?
165     inet_ntoa6(&T->dccp_connp->connua_v6addr.connua_laddr) : "<unknown>"
166     dccps_raddr = T ?
167     inet_ntoa6(&T->dccp_connp->connua_v6addr.connua_faddr) : "<unknown>"
168     dccps_state = T ? T->dccp_state : DCCP_STATE_CLOSED;
169     dccps_iss = T ? T->dccp_iss : 0;
170 };

172 /*
173 * Note: although we specify that the old state argument used as the
174 * input to the tcplsinfo_t translator is an int32_t, it reaches us as an
175 * int64_t (since it is a probe argument) so explicitly cast it back to
176 * interpret the negatively-valued states correctly.
177 */
178 #pragma D binding "1.10" translator
179 translator dccplsinfo_t < int64_t I > {
180     dccps_state = (int32_t) I;
181 };
182 #endif /* ! codereview */

```

new/usr/src/lib/libdtrace/common/dccp.sed.in

1

1289 Sat Aug 18 10:37:03 2012

new/usr/src/lib/libdtrace/common/dccp.sed.in

dccp: basic dtrace

```
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 #include <inet/dccp.h>
26 #include <sys/netstack.h>

28 #define SED_REPLACE(x)  s/#x/x/g

31 SED_REPLACE(DCCPS_CLOSED)
32 SED_REPLACE(DCCPS_BOUND)
33 SED_REPLACE(DCCPS_REQUEST)
34 SED_REPLACE(DCCPS_LISTEN)
35 SED_REPLACE(DCCPS_PARTOPEN)
36 SED_REPLACE(DCCPS_RESPOND)
37 SED_REPLACE(DCCPS_OPEN)
38 SED_REPLACE(DCCPS_CLOSING)
39 SED_REPLACE(DCCPS_CLOSEREQ)
40 SED_REPLACE(DCCPS_TIMEWAIT)

42 SED_REPLACE(DCCP_MIN_HEADER_LENGTH)
43 #endif /* ! codereview */
```

```
*****
53649 Sat Aug 18 10:37:03 2012
new/usr/src/lib/libdtrace/common/dt_open.c
dccp: basic dtrace
*****
```

```
_____unchanged_portion_omitted_____
```

```
82 /*
83  * The version number should be increased for every customer visible release
84  * of Solaris. The major number should be incremented when a fundamental
85  * change has been made that would affect all consumers, and would reflect
86  * sweeping changes to DTrace or the D language. The minor number should be
87  * incremented when a change is introduced that could break scripts that had
88  * previously worked; for example, adding a new built-in variable could break
89  * a script which was already using that identifier. The micro number should
90  * be changed when introducing functionality changes or major bug fixes that
91  * do not affect backward compatibility -- this is merely to make capabilities
92  * easily determined from the version number. Minor bugs do not require any
93  * modification to the version number.
94  */
95 #define DT_VERS_1_0      DT_VERSION_NUMBER(1, 0, 0)
96 #define DT_VERS_1_1      DT_VERSION_NUMBER(1, 1, 0)
97 #define DT_VERS_1_2      DT_VERSION_NUMBER(1, 2, 0)
98 #define DT_VERS_1_2_1    DT_VERSION_NUMBER(1, 2, 1)
99 #define DT_VERS_1_2_2    DT_VERSION_NUMBER(1, 2, 2)
100 #define DT_VERS_1_3      DT_VERSION_NUMBER(1, 3, 0)
101 #define DT_VERS_1_4      DT_VERSION_NUMBER(1, 4, 0)
102 #define DT_VERS_1_4_1    DT_VERSION_NUMBER(1, 4, 1)
103 #define DT_VERS_1_5      DT_VERSION_NUMBER(1, 5, 0)
104 #define DT_VERS_1_6      DT_VERSION_NUMBER(1, 6, 0)
105 #define DT_VERS_1_6_1    DT_VERSION_NUMBER(1, 6, 1)
106 #define DT_VERS_1_6_2    DT_VERSION_NUMBER(1, 6, 2)
107 #define DT_VERS_1_6_3    DT_VERSION_NUMBER(1, 6, 3)
108 #define DT_VERS_1_7      DT_VERSION_NUMBER(1, 7, 0)
109 #define DT_VERS_1_7_1    DT_VERSION_NUMBER(1, 7, 1)
110 #define DT_VERS_1_8      DT_VERSION_NUMBER(1, 8, 0)
111 #define DT_VERS_1_8_1    DT_VERSION_NUMBER(1, 8, 1)
112 #define DT_VERS_1_9      DT_VERSION_NUMBER(1, 9, 0)
113 #define DT_VERS_1_10     DT_VERSION_NUMBER(1, 10, 0)
114 #define DT_VERS_LATEST   DT_VERS_1_10
115 #define DT_VERS_STRING   "Sun D 1.10"
116 #define DT_VERS_LATEST   DT_VERS_1_9
117 #define DT_VERS_STRING   "Sun D 1.9"
```

```
117 const dt_version_t dttrace_versions[] = {
118     DT_VERS_1_0, /* D API 1.0.0 (PSARC 2001/466) Solaris 10 FCS */
119     DT_VERS_1_1, /* D API 1.1.0 Solaris Express 6/05 */
120     DT_VERS_1_2, /* D API 1.2.0 Solaris 10 Update 1 */
121     DT_VERS_1_2_1, /* D API 1.2.1 Solaris Express 4/06 */
122     DT_VERS_1_2_2, /* D API 1.2.2 Solaris Express 6/06 */
123     DT_VERS_1_3, /* D API 1.3 Solaris Express 10/06 */
124     DT_VERS_1_4, /* D API 1.4 Solaris Express 2/07 */
125     DT_VERS_1_4_1, /* D API 1.4.1 Solaris Express 4/07 */
126     DT_VERS_1_5, /* D API 1.5 Solaris Express 7/07 */
127     DT_VERS_1_6, /* D API 1.6 */
128     DT_VERS_1_6_1, /* D API 1.6.1 */
129     DT_VERS_1_6_2, /* D API 1.6.2 */
130     DT_VERS_1_6_3, /* D API 1.6.3 */
131     DT_VERS_1_7, /* D API 1.7 */
132     DT_VERS_1_7_1, /* D API 1.7.1 */
133     DT_VERS_1_8, /* D API 1.8 */
134     DT_VERS_1_8_1, /* D API 1.8.1 */
135     DT_VERS_1_9, /* D API 1.9 */
136     DT_VERS_1_10, /* D API 1.10 */
137 #endif /* ! codereview */
138     0
```

```
139 };
```

```
141 /*
142  * Table of global identifiers. This is used to populate the global identifier
143  * hash when a new dtrace client open occurs. For more info see dt_ident.h.
144  * The global identifiers that represent functions use the dt_idops_func ops
145  * and specify the private data pointer as a prototype string which is parsed
146  * when the identifier is first encountered. These prototypes look like ANSI
147  * C function prototypes except that the special symbol "@" can be used as a
148  * wildcard to represent a single parameter of any type (i.e. any dt_node_t).
149  * The standard "..." notation can also be used to represent varargs. An empty
150  * parameter list is taken to mean void (that is, no arguments are permitted).
151  * A parameter enclosed in square brackets (e.g. "[int]") denotes an optional
152  * argument.
153  */
154 static const dt_ident_t dttrace_globals[] = {
155     { "alloca", DT_IDENT_FUNC, 0, DIF_SUBR_ALLOCA, DT_ATTR_STABCMN, DT_VERS_1_0,
156       &dt_idops_func, "void *(size_t)" },
157     { "arg0", DT_IDENT_SCALAR, 0, DIF_VAR_ARG0, DT_ATTR_STABCMN, DT_VERS_1_0,
158       &dt_idops_type, "int64_t" },
159     { "arg1", DT_IDENT_SCALAR, 0, DIF_VAR_ARG1, DT_ATTR_STABCMN, DT_VERS_1_0,
160       &dt_idops_type, "int64_t" },
161     { "arg2", DT_IDENT_SCALAR, 0, DIF_VAR_ARG2, DT_ATTR_STABCMN, DT_VERS_1_0,
162       &dt_idops_type, "int64_t" },
163     { "arg3", DT_IDENT_SCALAR, 0, DIF_VAR_ARG3, DT_ATTR_STABCMN, DT_VERS_1_0,
164       &dt_idops_type, "int64_t" },
165     { "arg4", DT_IDENT_SCALAR, 0, DIF_VAR_ARG4, DT_ATTR_STABCMN, DT_VERS_1_0,
166       &dt_idops_type, "int64_t" },
167     { "arg5", DT_IDENT_SCALAR, 0, DIF_VAR_ARG5, DT_ATTR_STABCMN, DT_VERS_1_0,
168       &dt_idops_type, "int64_t" },
169     { "arg6", DT_IDENT_SCALAR, 0, DIF_VAR_ARG6, DT_ATTR_STABCMN, DT_VERS_1_0,
170       &dt_idops_type, "int64_t" },
171     { "arg7", DT_IDENT_SCALAR, 0, DIF_VAR_ARG7, DT_ATTR_STABCMN, DT_VERS_1_0,
172       &dt_idops_type, "int64_t" },
173     { "arg8", DT_IDENT_SCALAR, 0, DIF_VAR_ARG8, DT_ATTR_STABCMN, DT_VERS_1_0,
174       &dt_idops_type, "int64_t" },
175     { "arg9", DT_IDENT_SCALAR, 0, DIF_VAR_ARG9, DT_ATTR_STABCMN, DT_VERS_1_0,
176       &dt_idops_type, "int64_t" },
177     { "args", DT_IDENT_ARRAY, 0, DIF_VAR_ARGS, DT_ATTR_STABCMN, DT_VERS_1_0,
178       &dt_idops_args, NULL },
179     { "avg", DT_IDENT_AGGFUNC, 0, DTRACEAGG_AVG, DT_ATTR_STABCMN, DT_VERS_1_0,
180       &dt_idops_func, "void(@)" },
181     { "basename", DT_IDENT_FUNC, 0, DIF_SUBR_BASENAME, DT_ATTR_STABCMN, DT_VERS_1_0,
182       &dt_idops_func, "string(const char *)" },
183     { "bcopy", DT_IDENT_FUNC, 0, DIF_SUBR_BCOPY, DT_ATTR_STABCMN, DT_VERS_1_0,
184       &dt_idops_func, "void(void *, void *, size_t)" },
185     { "breakpoint", DT_IDENT_ACTFUNC, 0, DT_ACT_BREAKPOINT,
186       DT_ATTR_STABCMN, DT_VERS_1_0,
187       &dt_idops_func, "void()" },
188     { "caller", DT_IDENT_SCALAR, 0, DIF_VAR_CALLER, DT_ATTR_STABCMN, DT_VERS_1_0,
189       &dt_idops_type, "uintptr_t" },
190     { "chill", DT_IDENT_ACTFUNC, 0, DT_ACT_CHILL, DT_ATTR_STABCMN, DT_VERS_1_0,
191       &dt_idops_func, "void(int)" },
192     { "cleanpath", DT_IDENT_FUNC, 0, DIF_SUBR_CLEANPATH, DT_ATTR_STABCMN,
193       DT_VERS_1_0, &dt_idops_func, "string(const char *)" },
194     { "clear", DT_IDENT_ACTFUNC, 0, DT_ACT_CLEAR, DT_ATTR_STABCMN, DT_VERS_1_0,
195       &dt_idops_func, "void(...)" },
196     { "commit", DT_IDENT_ACTFUNC, 0, DT_ACT_COMMIT, DT_ATTR_STABCMN, DT_VERS_1_0,
197       &dt_idops_func, "void(int)" },
198     { "copyin", DT_IDENT_FUNC, 0, DIF_SUBR_COPYIN, DT_ATTR_STABCMN, DT_VERS_1_0,
199       &dt_idops_func, "void *(uintptr_t, size_t)" },
200     { "copyinstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINSTR,
201       DT_ATTR_STABCMN, DT_VERS_1_0,
202       &dt_idops_func, "string(uintptr_t, [size_t])" },
203     { "copyinto", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINTO, DT_ATTR_STABCMN,
204       DT_VERS_1_0, &dt_idops_func, "void(uintptr_t, size_t, void *)" },
```



```

205 { "copyout", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUT, DT_ATTR_STABCMN, DT_VERS_1_0,
206     &dt_idops_func, "void(void *, uintptr_t, size_t)" },
207 { "copyoutstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUTSTR,
208     DT_ATTR_STABCMN, DT_VERS_1_0,
209     &dt_idops_func, "void(char *, uintptr_t, size_t)" },
210 { "count", DT_IDENT_AGGFUNC, 0, DTRACEAGG_COUNT, DT_ATTR_STABCMN, DT_VERS_1_0,
211     &dt_idops_func, "void()" },
212 { "curthread", DT_IDENT_SCALAR, 0, DIF_VAR_CURTHREAD,
213     { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_PRIVATE,
214       DTRACE_CLASS_COMMON }, DT_VERS_1_0,
215     &dt_idops_type, "genunix'kthread_t *" },
216 { "ddi_pathname", DT_IDENT_FUNC, 0, DIF_SUBR_DDI_PATHNAME,
217     DT_ATTR_EVOLCMN, DT_VERS_1_0,
218     &dt_idops_func, "string(void *, int64_t)" },
219 { "denormalize", DT_IDENT_ACTFUNC, 0, DT_ACT_DENORMALIZE, DT_ATTR_STABCMN,
220     DT_VERS_1_0, &dt_idops_func, "void(...)" },
221 { "dirname", DT_IDENT_FUNC, 0, DIF_SUBR_DIRNAME, DT_ATTR_STABCMN, DT_VERS_1_0,
222     &dt_idops_func, "string(const char *)" },
223 { "discard", DT_IDENT_ACTFUNC, 0, DT_ACT_DISCARD, DT_ATTR_STABCMN, DT_VERS_1_0,
224     &dt_idops_func, "void(int)" },
225 { "epid", DT_IDENT_SCALAR, 0, DIF_VAR_EPID, DT_ATTR_STABCMN, DT_VERS_1_0,
226     &dt_idops_type, "uint_t" },
227 { "errno", DT_IDENT_SCALAR, 0, DIF_VAR_ERRNO, DT_ATTR_STABCMN, DT_VERS_1_0,
228     &dt_idops_type, "int" },
229 { "execname", DT_IDENT_SCALAR, 0, DIF_VAR_EXECNAME,
230     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
231 { "exit", DT_IDENT_ACTFUNC, 0, DT_ACT_EXIT, DT_ATTR_STABCMN, DT_VERS_1_0,
232     &dt_idops_func, "void(int)" },
233 { "freopen", DT_IDENT_ACTFUNC, 0, DT_ACT_FREOPEN, DT_ATTR_STABCMN,
234     DT_VERS_1_1, &dt_idops_func, "void(@, ...)" },
235 { "ftruncate", DT_IDENT_ACTFUNC, 0, DT_ACT_FTRUNCATE, DT_ATTR_STABCMN,
236     DT_VERS_1_0, &dt_idops_func, "void()" },
237 { "func", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
238     DT_VERS_1_2, &dt_idops_func, "symaddr(uintptr_t)" },
239 { "getmajor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMAJOR,
240     DT_ATTR_EVOLCMN, DT_VERS_1_0,
241     &dt_idops_func, "genunix'major_t(genunix'dev_t)" },
242 { "getminor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMINOR,
243     DT_ATTR_EVOLCMN, DT_VERS_1_0,
244     &dt_idops_func, "genunix'minor_t(genunix'dev_t)" },
245 { "htonl", DT_IDENT_FUNC, 0, DIF_SUBR_HTONL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
246     &dt_idops_func, "uint32_t(uint32_t)" },
247 { "htonll", DT_IDENT_FUNC, 0, DIF_SUBR_HTONLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
248     &dt_idops_func, "uint64_t(uint64_t)" },
249 { "htons", DT_IDENT_FUNC, 0, DIF_SUBR_HTONS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
250     &dt_idops_func, "uint16_t(uint16_t)" },
251 { "gid", DT_IDENT_SCALAR, 0, DIF_VAR_GID, DT_ATTR_STABCMN, DT_VERS_1_0,
252     &dt_idops_type, "gid_t" },
253 { "id", DT_IDENT_SCALAR, 0, DIF_VAR_ID, DT_ATTR_STABCMN, DT_VERS_1_0,
254     &dt_idops_type, "uint_t" },
255 { "index", DT_IDENT_FUNC, 0, DIF_SUBR_INDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
256     &dt_idops_func, "int(const char *, const char *, [int])" },
257 { "inet_ntoa", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA, DT_ATTR_STABCMN,
258     DT_VERS_1_5, &dt_idops_func, "string(ipaddr_t *)" },
259 { "inet_ntoa6", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA6, DT_ATTR_STABCMN,
260     DT_VERS_1_5, &dt_idops_func, "string(in6_addr_t *)" },
261 { "inet_ntop", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOP, DT_ATTR_STABCMN,
262     DT_VERS_1_5, &dt_idops_func, "string(int, void *)" },
263 { "ipl", DT_IDENT_SCALAR, 0, DIF_VAR_IPL, DT_ATTR_STABCMN, DT_VERS_1_0,
264     &dt_idops_type, "uint_t" },
265 { "jstack", DT_IDENT_ACTFUNC, 0, DT_ACT_JSTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
266     &dt_idops_func, "stack(...)" },
267 { "lltost", DT_IDENT_FUNC, 0, DIF_SUBR_LLTOST, DT_ATTR_STABCMN, DT_VERS_1_0,
268     &dt_idops_func, "string(int64_t, [int])" },
269 { "llquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_LLQUANTIZE, DT_ATTR_STABCMN,
270     DT_VERS_1_7, &dt_idops_func,

```

```

271     "void(@, int32_t, int32_t, int32_t, ...)" },
272 { "lquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_IQUANTIZE,
273     DT_ATTR_STABCMN, DT_VERS_1_0,
274     &dt_idops_func, "void(@, int32_t, int32_t, ...)" },
275 { "max", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MAX, DT_ATTR_STABCMN, DT_VERS_1_0,
276     &dt_idops_func, "void(@)" },
277 { "min", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MIN, DT_ATTR_STABCMN, DT_VERS_1_0,
278     &dt_idops_func, "void(@)" },
279 { "mod", DT_IDENT_ACTFUNC, 0, DT_ACT_MOD, DT_ATTR_STABCMN,
280     DT_VERS_1_2, &dt_idops_func, "symaddr(uintptr_t)" },
281 { "msgdsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGDSIZE,
282     DT_ATTR_STABCMN, DT_VERS_1_0,
283     &dt_idops_func, "size_t(mblk_t *)" },
284 { "msgsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGSIZE,
285     DT_ATTR_STABCMN, DT_VERS_1_0,
286     &dt_idops_func, "size_t(mblk_t *)" },
287 { "mutex_owned", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNED,
288     DT_ATTR_EVOLCMN, DT_VERS_1_0,
289     &dt_idops_func, "int(genunix'kmutex_t *)" },
290 { "mutex_owner", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNER,
291     DT_ATTR_EVOLCMN, DT_VERS_1_0,
292     &dt_idops_func, "genunix'kthread_t *(genunix'kmutex_t *)" },
293 { "mutex_type_adaptive", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_ADAPTIVE,
294     DT_ATTR_EVOLCMN, DT_VERS_1_0,
295     &dt_idops_func, "int(genunix'kmutex_t *)" },
296 { "mutex_type_spin", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_SPIN,
297     DT_ATTR_EVOLCMN, DT_VERS_1_0,
298     &dt_idops_func, "int(genunix'kmutex_t *)" },
299 { "ntohl", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
300     &dt_idops_func, "uint32_t(uint32_t)" },
301 { "ntohl1", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
302     &dt_idops_func, "uint64_t(uint64_t)" },
303 { "ntohs", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
304     &dt_idops_func, "uint16_t(uint16_t)" },
305 { "normalize", DT_IDENT_ACTFUNC, 0, DT_ACT_NORMALIZE, DT_ATTR_STABCMN,
306     DT_VERS_1_0, &dt_idops_func, "void(...)" },
307 { "panic", DT_IDENT_ACTFUNC, 0, DT_ACT_PANIC, DT_ATTR_STABCMN, DT_VERS_1_0,
308     &dt_idops_func, "void()" },
309 { "pid", DT_IDENT_SCALAR, 0, DIF_VAR_PID, DT_ATTR_STABCMN, DT_VERS_1_0,
310     &dt_idops_type, "pid_t" },
311 { "ppid", DT_IDENT_SCALAR, 0, DIF_VAR_PPID, DT_ATTR_STABCMN, DT_VERS_1_0,
312     &dt_idops_type, "pid_t" },
313 { "print", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINT, DT_ATTR_STABCMN, DT_VERS_1_9,
314     &dt_idops_func, "void(@)" },
315 { "printa", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTA, DT_ATTR_STABCMN, DT_VERS_1_0,
316     &dt_idops_func, "void(@, ...)" },
317 { "printf", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTF, DT_ATTR_STABCMN, DT_VERS_1_0,
318     &dt_idops_func, "void(@, ...)" },
319 { "probefunc", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEFUNC,
320     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
321 { "probemod", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEMOD,
322     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
323 { "probenam", DT_IDENT_SCALAR, 0, DIF_VAR_PROBENAME,
324     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
325 { "probeprov", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEPROV,
326     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
327 { "progenyof", DT_IDENT_FUNC, 0, DIF_SUBR_PROGENYOOF,
328     DT_ATTR_STABCMN, DT_VERS_1_0,
329     &dt_idops_func, "int(pid_t)" },
330 { "quantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_QUANTIZE,
331     DT_ATTR_STABCMN, DT_VERS_1_0,
332     &dt_idops_func, "void(@, ...)" },
333 { "raise", DT_IDENT_ACTFUNC, 0, DT_ACT_RAISE, DT_ATTR_STABCMN, DT_VERS_1_0,
334     &dt_idops_func, "void(int)" },
335 { "rand", DT_IDENT_FUNC, 0, DIF_SUBR_RAND, DT_ATTR_STABCMN, DT_VERS_1_0,
336     &dt_idops_func, "int()" },

```

```

337 { "rindex", DT_IDENT_FUNC, 0, DIF_SUBR_RINDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
338     &dt_idops_func, "int(const char *, const char *, [int])" },
339 { "rw_iswriter", DT_IDENT_FUNC, 0, DIF_SUBR_RW_ISWRITER,
340     DT_ATTR_EVOLCMN, DT_VERS_1_0,
341     &dt_idops_func, "int(genunix'krwlock_t *)" },
342 { "rw_read_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_READ_HELD,
343     DT_ATTR_EVOLCMN, DT_VERS_1_0,
344     &dt_idops_func, "int(genunix'krwlock_t *)" },
345 { "rw_write_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_WRITE_HELD,
346     DT_ATTR_EVOLCMN, DT_VERS_1_0,
347     &dt_idops_func, "int(genunix'krwlock_t *)" },
348 { "self", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
349     &dt_idops_type, "void" },
350 { "setopt", DT_IDENT_ACTFUNC, 0, DT_ACT_SETOPT, DT_ATTR_STABCMN,
351     DT_VERS_1_2, &dt_idops_func, "void(const char *, [const char *])" },
352 { "speculate", DT_IDENT_ACTFUNC, 0, DT_ACT_SPECULATE,
353     DT_ATTR_STABCMN, DT_VERS_1_0,
354     &dt_idops_func, "void(int)" },
355 { "speculation", DT_IDENT_FUNC, 0, DIF_SUBR_SPECULATION,
356     DT_ATTR_STABCMN, DT_VERS_1_0,
357     &dt_idops_func, "int()" },
358 { "stack", DT_IDENT_ACTFUNC, 0, DT_ACT_STACK, DT_ATTR_STABCMN, DT_VERS_1_0,
359     &dt_idops_func, "stack(...)" },
360 { "stackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_STACKDEPTH,
361     DT_ATTR_STABCMN, DT_VERS_1_0,
362     &dt_idops_type, "uint32_t" },
363 { "stddev", DT_IDENT_AGGFUNC, 0, DTRACEAGG_STDDEV, DT_ATTR_STABCMN,
364     DT_VERS_1_6, &dt_idops_func, "void(@)" },
365 { "stop", DT_IDENT_ACTFUNC, 0, DT_ACT_STOP, DT_ATTR_STABCMN, DT_VERS_1_0,
366     &dt_idops_func, "void()" },
367 { "strchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
368     &dt_idops_func, "string(const char *, char)" },
369 { "strlen", DT_IDENT_FUNC, 0, DIF_SUBR_STRLEN, DT_ATTR_STABCMN, DT_VERS_1_0,
370     &dt_idops_func, "size_t(const char *)" },
371 { "strjoin", DT_IDENT_FUNC, 0, DIF_SUBR_STRJOIN, DT_ATTR_STABCMN, DT_VERS_1_0,
372     &dt_idops_func, "string(const char *, const char *)" },
373 { "strrchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
374     &dt_idops_func, "string(const char *, char)" },
375 { "strstr", DT_IDENT_FUNC, 0, DIF_SUBR_STRSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
376     &dt_idops_func, "string(const char *, const char *)" },
377 { "strtok", DT_IDENT_FUNC, 0, DIF_SUBR_STRTOK, DT_ATTR_STABCMN, DT_VERS_1_1,
378     &dt_idops_func, "string(const char *, const char *)" },
379 { "substr", DT_IDENT_FUNC, 0, DIF_SUBR_SUBSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
380     &dt_idops_func, "string(const char *, int, [int])" },
381 { "sum", DT_IDENT_AGGFUNC, 0, DTRACEAGG_SUM, DT_ATTR_STABCMN, DT_VERS_1_0,
382     &dt_idops_func, "void(@)" },
383 { "sym", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
384     DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
385 { "system", DT_IDENT_ACTFUNC, 0, DT_ACT_SYSTEM, DT_ATTR_STABCMN, DT_VERS_1_0,
386     &dt_idops_func, "void(@, ...)" },
387 { "this", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
388     &dt_idops_type, "void" },
389 { "tid", DT_IDENT_SCALAR, 0, DIF_VAR_TID, DT_ATTR_STABCMN, DT_VERS_1_0,
390     &dt_idops_type, "id_t" },
391 { "timestamp", DT_IDENT_SCALAR, 0, DIF_VAR_TIMESTAMP,
392     DT_ATTR_STABCMN, DT_VERS_1_0,
393     &dt_idops_type, "uint64_t" },
394 { "tolower", DT_IDENT_FUNC, 0, DIF_SUBR_TOLOWER, DT_ATTR_STABCMN, DT_VERS_1_8,
395     &dt_idops_func, "string(const char *)" },
396 { "toupper", DT_IDENT_FUNC, 0, DIF_SUBR_TOUPPER, DT_ATTR_STABCMN, DT_VERS_1_8,
397     &dt_idops_func, "string(const char *)" },
398 { "trace", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACE, DT_ATTR_STABCMN, DT_VERS_1_0,
399     &dt_idops_func, "void(@)" },
400 { "tracemem", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACEMEM,
401     DT_ATTR_STABCMN, DT_VERS_1_0,
402     &dt_idops_func, "void(@, size_t, ...)" },

```

```

403 { "trunc", DT_IDENT_ACTFUNC, 0, DT_ACT_TRUNC, DT_ATTR_STABCMN,
404     DT_VERS_1_0, &dt_idops_func, "void(...)" },
405 { "uaddr", DT_IDENT_ACTFUNC, 0, DT_ACT_UADDR, DT_ATTR_STABCMN,
406     DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
407 { "ucaller", DT_IDENT_SCALAR, 0, DIF_VAR_UCALLER, DT_ATTR_STABCMN,
408     DT_VERS_1_2, &dt_idops_type, "uint64_t" },
409 { "ufunc", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
410     DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
411 { "uid", DT_IDENT_SCALAR, 0, DIF_VAR_UID, DT_ATTR_STABCMN, DT_VERS_1_0,
412     &dt_idops_type, "uid_t" },
413 { "umod", DT_IDENT_ACTFUNC, 0, DT_ACT_UMOD, DT_ATTR_STABCMN,
414     DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
415 { "uregs", DT_IDENT_ARRAY, 0, DIF_VAR_UREGS, DT_ATTR_STABCMN, DT_VERS_1_0,
416     &dt_idops_regs, NULL },
417 { "ustack", DT_IDENT_ACTFUNC, 0, DT_ACT_USTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
418     &dt_idops_func, "stack(...)" },
419 { "ustackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_USTACKDEPTH,
420     DT_ATTR_STABCMN, DT_VERS_1_2,
421     &dt_idops_type, "uint32_t" },
422 { "usym", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
423     DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
424 { "vmregs", DT_IDENT_ARRAY, 0, DIF_VAR_VMREGS, DT_ATTR_STABCMN, DT_VERS_1_7,
425     &dt_idops_regs, NULL },
426 { "vtimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_VTIMESTAMP,
427     DT_ATTR_STABCMN, DT_VERS_1_0,
428     &dt_idops_type, "uint64_t" },
429 { "walltimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_WALLTIMESTAMP,
430     DT_ATTR_STABCMN, DT_VERS_1_0,
431     &dt_idops_type, "int64_t" },
432 { "zonename", DT_IDENT_SCALAR, 0, DIF_VAR_ZONENAME,
433     DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
434 { NULL, 0, 0, 0, { 0, 0, 0 }, 0, NULL, NULL }
435 };
436
437 /*
438  * Tables of ILP32 intrinsic integer and floating-point type templates to use
439  * to populate the dynamic "C" CTF type container.
440  */
441 static const dt_intrinsic_t dttrace_intrinsics_32[] = {
442     { "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
443     { "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
444     { "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
445     { "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
446     { "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
447     { "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
448     { "long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
449     { "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
450     { "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
451     { "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
452     { "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
453     { "signed long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
454     { "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
455     { "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
456     { "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
457     { "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
458     { "unsigned long", { 0, 0, 32 }, CTF_K_INTEGER },
459     { "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
460     { "Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
461     { "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
462     { "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
463     { "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
464     { "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
465     { "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
466     { "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
467     { "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
468     { "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },

```

```

469 { "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
470 { NULL, { 0, 0, 0 }, 0 }
471 };

473 /*
474  * Tables of LP64 intrinsic integer and floating-point type templates to use
475  * to populate the dynamic "C" CTF type container.
476  */
477 static const dt_intrinsic_t _dtrace_intrinsics_64[] = {
478 { "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
479 { "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
480 { "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
481 { "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
482 { "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
483 { "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
484 { "long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
485 { "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
486 { "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
487 { "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
488 { "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
489 { "signed long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
490 { "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
491 { "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
492 { "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
493 { "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
494 { "unsigned long", { 0, 0, 64 }, CTF_K_INTEGER },
495 { "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
496 { "Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
497 { "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
498 { "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
499 { "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
500 { "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
501 { "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
502 { "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
503 { "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
504 { "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },
505 { "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
506 { NULL, { 0, 0, 0 }, 0 }
507 };

509 /*
510  * Tables of ILP32 typedefs to use to populate the dynamic "D" CTF container.
511  * These aliases ensure that D definitions can use typical <sys/types.h> names.
512  */
513 static const dt_typedef_t _dtrace_typedefs_32[] = {
514 { "char", "int8_t" },
515 { "short", "int16_t" },
516 { "int", "int32_t" },
517 { "long long", "int64_t" },
518 { "int", "intptr_t" },
519 { "int", "ssize_t" },
520 { "unsigned char", "uint8_t" },
521 { "unsigned short", "uint16_t" },
522 { "unsigned", "uint32_t" },
523 { "unsigned long long", "uint64_t" },
524 { "unsigned char", "uchar_t" },
525 { "unsigned short", "ushort_t" },
526 { "unsigned", "uint_t" },
527 { "unsigned long", "ulong_t" },
528 { "unsigned long long", "u_longlong_t" },
529 { "int", "ptrdiff_t" },
530 { "unsigned", "uintptr_t" },
531 { "unsigned", "size_t" },
532 { "long", "id_t" },
533 { "long", "pid_t" },
534 { NULL, NULL }

```

```

535 };

537 /*
538  * Tables of LP64 typedefs to use to populate the dynamic "D" CTF container.
539  * These aliases ensure that D definitions can use typical <sys/types.h> names.
540  */
541 static const dt_typedef_t _dtrace_typedefs_64[] = {
542 { "char", "int8_t" },
543 { "short", "int16_t" },
544 { "int", "int32_t" },
545 { "long", "int64_t" },
546 { "long", "intptr_t" },
547 { "long", "ssize_t" },
548 { "unsigned char", "uint8_t" },
549 { "unsigned short", "uint16_t" },
550 { "unsigned", "uint32_t" },
551 { "unsigned long", "uint64_t" },
552 { "unsigned char", "uchar_t" },
553 { "unsigned short", "ushort_t" },
554 { "unsigned", "uint_t" },
555 { "unsigned long", "ulong_t" },
556 { "unsigned long long", "u_longlong_t" },
557 { "long", "ptrdiff_t" },
558 { "unsigned long", "uintptr_t" },
559 { "unsigned long", "size_t" },
560 { "int", "id_t" },
561 { "int", "pid_t" },
562 { NULL, NULL }
563 };

565 /*
566  * Tables of ILP32 integer type templates used to populate the.dtp->dt_ints[]
567  * cache when a new dtrace client open occurs. Values are set by dtrace_open().
568  */
569 static const dt_intdesc_t _dtrace_ints_32[] = {
570 { "int", NULL, CTF_ERR, 0x7fffffffULL },
571 { "unsigned int", NULL, CTF_ERR, 0xffffffffULL },
572 { "long", NULL, CTF_ERR, 0x7fffffffULL },
573 { "unsigned long", NULL, CTF_ERR, 0xffffffffULL },
574 { "long long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
575 { "unsigned long long", NULL, CTF_ERR, 0xffffffffffffffffULL }
576 };

578 /*
579  * Tables of LP64 integer type templates used to populate the.dtp->dt_ints[]
580  * cache when a new dtrace client open occurs. Values are set by dtrace_open().
581  */
582 static const dt_intdesc_t _dtrace_ints_64[] = {
583 { "int", NULL, CTF_ERR, 0x7fffffffULL },
584 { "unsigned int", NULL, CTF_ERR, 0xffffffffULL },
585 { "long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
586 { "unsigned long", NULL, CTF_ERR, 0xffffffffffffffffULL },
587 { "long long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
588 { "unsigned long long", NULL, CTF_ERR, 0xffffffffffffffffULL }
589 };

591 /*
592  * Table of macro variable templates used to populate the macro identifier hash
593  * when a new dtrace client open occurs. Values are set by dtrace_update().
594  */
595 static const dt_ident_t _dtrace_macros[] = {
596 { "egid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
597 { "euid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
598 { "gid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
599 { "pid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
600 { "pgid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },

```

```

601 { "ppid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
602 { "projid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
603 { "sid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
604 { "taskid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
605 { "target", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
606 { "uid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
607 { NULL, 0, 0, 0, { 0, 0, 0 }, 0 }
608 };

610 /*
611  * Hard-wired definition string to be compiled and cached every time a new
612  * DTrace library handle is initialized. This string should only be used to
613  * contain definitions that should be present regardless of DTRACE_O_NOLIBS.
614  */
615 static const char _dtrace_hardwire[] = "\n
616 inline long NULL = 0; \n\
617 #pragma D binding \"1.0\" NULL\n\
618 ";

620 /*
621  * Default DTrace configuration to use when opening libdtrace DTRACE_O_NODEV.
622  * If DTRACE_O_NODEV is not set, we load the configuration from the kernel.
623  * The use of CTF_MODEL_NATIVE is more subtle than it might appear: we are
624  * relying on the fact that when running dtrace(1M), isaexec will invoke the
625  * binary with the same bitness as the kernel, which is what we want by default
626  * when generating our DIF. The user can override the choice using oflags.
627  */
628 static const dtrace_conf_t _dtrace_conf = {
629     DIF_VERSION,          /* dtc_difversion */
630     DIF_DIR_NREGS,        /* dtc_difintregs */
631     DIF_DTR_NREGS,        /* dtc_diftupregs */
632     CTF_MODEL_NATIVE      /* dtc_ctfmodel */
633 };

635 const dtrace_attribute_t _dtrace_maxattr = {
636     DTRACE_STABILITY_MAX,
637     DTRACE_STABILITY_MAX,
638     DTRACE_CLASS_MAX
639 };

641 const dtrace_attribute_t _dtrace_defattr = {
642     DTRACE_STABILITY_STABLE,
643     DTRACE_STABILITY_STABLE,
644     DTRACE_CLASS_COMMON
645 };

647 const dtrace_attribute_t _dtrace_symattr = {
648     DTRACE_STABILITY_PRIVATE,
649     DTRACE_STABILITY_PRIVATE,
650     DTRACE_CLASS_UNKNOWN
651 };

653 const dtrace_attribute_t _dtrace_typattr = {
654     DTRACE_STABILITY_PRIVATE,
655     DTRACE_STABILITY_PRIVATE,
656     DTRACE_CLASS_UNKNOWN
657 };

659 const dtrace_attribute_t _dtrace_prvattr = {
660     DTRACE_STABILITY_PRIVATE,
661     DTRACE_STABILITY_PRIVATE,
662     DTRACE_CLASS_UNKNOWN
663 };

665 const dtrace_pattn_t _dtrace_prvdesc = {
666     { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },

```

```

667 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
668 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
669 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
670 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
671 };

673 const char *_dtrace_defcpp = "/usr/ccs/lib/cpp"; /* default cpp(1) to invoke */
674 const char *_dtrace_defld = "/usr/ccs/bin/ld"; /* default ld(1) to invoke */

676 const char *_dtrace_libdir = "/usr/lib/dtrace"; /* default library directory */
677 const char *_dtrace_provdirdir = "/dev/dtrace/provider"; /* provider directory */

679 int _dtrace_strbuckets = 211; /* default number of hash buckets (prime) */
680 int _dtrace_intbuckets = 256; /* default number of integer buckets (Pof2) */
681 uint_t _dtrace_stsize = 256; /* default size of string intrinsic type */
682 uint_t _dtrace_stkindent = 14; /* default whitespace indent for stack/ustack */
683 uint_t _dtrace_pidbuckets = 64; /* default number of pid hash buckets */
684 uint_t _dtrace_pidlrulim = 8; /* default number of pid handles to cache */
685 size_t _dtrace_bufsize = 512; /* default dt_buf_create() size */
686 int _dtrace_argmax = 32; /* default maximum number of probe arguments */

688 int _dtrace_debug = 0; /* debug messages enabled (off) */
689 const char *const _dtrace_version = DT_VERS_STRING; /* API version string */
690 int _dtrace_rdvrs = RD_VERSION; /* rtld_db feature version */

692 typedef struct dt_fdlist {
693     int *df_fds; /* array of provider driver file descriptors */
694     uint_t df_ents; /* number of valid elements in df_fds[] */
695     uint_t df_size; /* size of df_fds[] */
696 } dt_fdlist_t;

698 #pragma init(_dtrace_init)
699 void
700 _dtrace_init(void)
701 {
702     _dtrace_debug = getenv("DTRACE_DEBUG") != NULL;

704     for (; _dtrace_rdvrs > 0; _dtrace_rdvrs--) {
705         if (rd_init(_dtrace_rdvrs) == RD_OK)
706             break;
707     }
708 }

710 static dtrace_hdl_t *
711 set_open_errno(dtrace_hdl_t *dtp, int *errp, int err)
712 {
713     if (dtp != NULL)
714         dtrace_close(dtp);
715     if (errp != NULL)
716         *errp = err;
717     return (NULL);
718 }

720 static void
721 dt_provmod_open(dt_provmod_t **provmod, dt_fdlist_t *dfp)
722 {
723     dt_provmod_t *prov;
724     char path[PATH_MAX];
725     struct dirent *dp, *ep;
726     DIR *dirp;
727     int fd;

729     if ((dirp = opendir(_dtrace_provdirdir)) == NULL)
730         return; /* failed to open directory; just skip it */

732     ep = alloca(sizeof (struct dirent) + PATH_MAX + 1);

```

```

733     bzero(ep, sizeof (struct dirent) + PATH_MAX + 1);

735     while (readdir_r(dirp, ep, &dp) == 0 && dp != NULL) {
736         if (dp->d_name[0] == '.')
737             continue; /* skip "." and ".." */

739         if (dfp->df_ents == dfp->df_size) {
740             uint_t size = dfp->df_size ? dfp->df_size * 2 : 16;
741             int *fds = realloc(dfp->df_fds, size * sizeof (int));

743             if (fds == NULL)
744                 break; /* skip the rest of this directory */

746             dfp->df_fds = fds;
747             dfp->df_size = size;
748         }

750         (void) snprintf(path, sizeof (path), "%s/%s",
751             _dtrace_providir, dp->d_name);

753         if ((fd = open(path, O_RDONLY)) == -1)
754             continue; /* failed to open driver; just skip it */

756         if (((prov = malloc(sizeof (dt_provm_t))) == NULL) ||
757             (prov->dp_name = malloc(strlen(dp->d_name) + 1)) == NULL) {
758             free(prov);
759             (void) close(fd);
760             break;
761         }

763         (void) strcpy(prov->dp_name, dp->d_name);
764         prov->dp_next = *provm;
765         *provm = prov;

767         dt_dprintf("opened provider %s\n", dp->d_name);
768         dfp->df_fds[dfp->df_ents++] = fd;
769     }

771     (void) closedir(dirp);
772 }

774 static void
775 dt_provm_destroy(dt_provm_t **provm)
776 {
777     dt_provm_t *next, *current;

779     for (current = *provm; current != NULL; current = next) {
780         next = current->dp_next;
781         free(current->dp_name);
782         free(current);
783     }

785     *provm = NULL;
786 }

788 static const char *
789 dt_get_sysinfo(int cmd, char *buf, size_t len)
790 {
791     ssize_t rv = sysinfo(cmd, buf, len);
792     char *p = buf;

794     if (rv < 0 || rv > len)
795         (void) snprintf(buf, len, "%s", "Unknown");

797     while ((p = strchr(p, '.')) != NULL)
798         *p++ = '_';

```

```

800     return (buf);
801 }

803 static dtrace_hdl_t *
804 dt_vopen(int version, int flags, int *errp,
805     const dtrace_vector_t *vector, void *arg)
806 {
807     dtrace_hdl_t *dtp = NULL;
808     int dtfd = -1, ftfd = -1, fterr = 0;
809     dtrace_prog_t *pgp;
810     dt_module_t *dmp;
811     dt_provm_t *provm = NULL;
812     int i, err;
813     struct rlimit rl;

815     const dt_intrinsic_t *dinp;
816     const dt_typedef_t *dtyp;
817     const dt_ident_t *idp;

819     dtrace_typeinfo_t dtt;
820     ctf_funcinfo_t ctc;
821     ctf_arinfo_t ctr;

823     dt_fdlist_t df = { NULL, 0, 0 };

825     char isadef[32], utsdef[32];
826     char s1[64], s2[64];

828     if (version <= 0)
829         return (set_open_errno(dtp, errp, EINVAL));

831     if (version > DTRACE_VERSION)
832         return (set_open_errno(dtp, errp, EDT_VERSION));

834     if (version < DTRACE_VERSION) {
835         /*
836          * Currently, increasing the library version number is used to
837          * denote a binary incompatible change. That is, a consumer
838          * of the library cannot run on a version of the library with
839          * a higher DTRACE_VERSION number than the consumer compiled
840          * against. Once the library API has been committed to,
841          * backwards binary compatibility will be required; at that
842          * time, this check should change to return EDT_OVERVERSION only
843          * if the specified version number is less than the version
844          * number at the time of interface commitment.
845          */
846         return (set_open_errno(dtp, errp, EDT_OVERVERSION));
847     }

849     if (flags & ~DTRACE_O_MASK)
850         return (set_open_errno(dtp, errp, EINVAL));

852     if ((flags & DTRACE_O_LP64) && (flags & DTRACE_O_ILP32))
853         return (set_open_errno(dtp, errp, EINVAL));

855     if (vector == NULL && arg != NULL)
856         return (set_open_errno(dtp, errp, EINVAL));

858     if (elf_version(EV_CURRENT) == EV_NONE)
859         return (set_open_errno(dtp, errp, EDT_ELFVERSION));

861     if (vector != NULL || (flags & DTRACE_O_NODEV))
862         goto alloc; /* do not attempt to open dtrace device */

864     /*

```

```

865  * Before we get going, crank our limit on file descriptors up to the
866  * hard limit. This is to allow for the fact that libproc keeps file
867  * descriptors to objects open for the lifetime of the proc handle;
868  * without raising our hard limit, we would have an acceptably small
869  * bound on the number of processes that we could concurrently
870  * instrument with the pid provider.
871  */
872  if (getrlimit(RLIMIT_NOFILE, &rl) == 0) {
873      rl.rlim_cur = rl.rlim_max;
874      (void) setrlimit(RLIMIT_NOFILE, &rl);
875  }

877  /*
878  * Get the device path of each of the providers. We hold them open
879  * in the df.df_fds list until we open the DTrace driver itself,
880  * allowing us to see all of the probes provided on this system. Once
881  * we have the DTrace driver open, we can safely close all the providers
882  * now that they have registered with the framework.
883  */
884  dt_provmode_open(&provmode, &df);

886  dtfd = open("/dev/dtrace/dtrace", O_RDWR);
887  err = errno; /* save errno from opening dtfd */

889  ftfd = open("/dev/dtrace/provider/fasttrap", O_RDWR);
890  fterr = ftfd == -1 ? errno : 0; /* save errno from open ftfd */

892  while (df.df_ents-- != 0)
893      (void) close(df.df_fds[df.df_ents]);

895  free(df.df_fds);

897  /*
898  * If we failed to open the dtrace device, fail dtrace_open().
899  * We convert some kernel errors to custom libdtrace errors to
900  * improve the resulting message from the usual strerror().
901  */
902  if (dtfd == -1) {
903      dt_provmode_destroy(&provmode);
904      switch (err) {
905          case ENOENT:
906              err = EDT_NOENT;
907              break;
908          case EBUSY:
909              err = EDT_BUSY;
910              break;
911          case EACCES:
912              err = EDT_ACCESS;
913              break;
914      }
915      return (set_open_errno(dtp, errp, err));
916  }

918  (void) fcntl(dtf, F_SETFD, FD_CLOEXEC);
919  (void) fcntl(ftfd, F_SETFD, FD_CLOEXEC);

921  alloc:
922  if ((dtp = malloc(sizeof (dtrace_hdl_t))) == NULL)
923      return (set_open_errno(dtp, errp, EDT_NOMEM));

925  bzero(dtp, sizeof (dtrace_hdl_t));
926  dtp->dt_oflags = flags;
927  dtp->dt_prcmode = DT_PROC_STOP_PREINIT;
928  dtp->dt_linkmode = DT_LINK_KERNEL;
929  dtp->dt_linktype = DT_LTYPE_ELF;
930  dtp->dt_xlatemode = DT_XL_STATIC;

```

```

931  dtp->dt_stdcmode = DT_STDC_XA;
932  dtp->dt_version = version;
933  dtp->dt_fd = dtfd;
934  dtp->dt_ftfd = ftfd;
935  dtp->dt_fterr = fterr;
936  dtp->dt_cdefs_fd = -1;
937  dtp->dt_ddefs_fd = -1;
938  dtp->dt_stdout_fd = -1;
939  dtp->dt_modbuckets = _dtrace_strbuckets;
940  dtp->dt_mods = calloc(dtp->dt_modbuckets, sizeof (dt_module_t *));
941  dtp->dt_provbuckets = _dtrace_strbuckets;
942  dtp->dt_provs = calloc(dtp->dt_provbuckets, sizeof (dt_provider_t *));
943  dt_proc_hash_create(dtp);
944  dtp->dt_vmax = DT_VERS_LATEST;
945  dtp->dt_cpp_path = strdup(_dtrace_defcpp);
946  dtp->dt_cpp_argv = malloc(sizeof (char *));
947  dtp->dt_cpp_argc = 1;
948  dtp->dt_cpp_args = 1;
949  dtp->dt_ld_path = strdup(_dtrace_defld);
950  dtp->dt_provmode = provmode;
951  dtp->dt_vector = vector;
952  dtp->dt_varg = arg;
953  dt_dof_init(dtp);
954  (void) uname(&dtp->dt_uts);

956  if (dtp->dt_mods == NULL || dtp->dt_provs == NULL ||
957      dtp->dt_procs == NULL || dtp->dt_ld_path == NULL ||
958      dtp->dt_cpp_path == NULL || dtp->dt_cpp_argv == NULL)
959      return (set_open_errno(dtp, errp, EDT_NOMEM));

961  for (i = 0; i < DTRACEOPT_MAX; i++)
962      dtp->dt_options[i] = DTRACEOPT_UNSET;

964  dtp->dt_cpp_argv[0] = (char *)strbasename(dtp->dt_cpp_path);

966  (void) snprintf(isadef, sizeof (isadef), "-D__SUNW_D_%u",
967      (uint_t)(sizeof (void *) * NBBY));

969  (void) snprintf(utsdef, sizeof (utsdef), "-D__%s_%s",
970      dt_get_sysinfo(SI_SYSNAME, s1, sizeof (s1)),
971      dt_get_sysinfo(SI_RELEASE, s2, sizeof (s2)));

973  if (dt_cpp_add_arg(dtp, "-D__sun") == NULL ||
974      dt_cpp_add_arg(dtp, "-D__unix") == NULL ||
975      dt_cpp_add_arg(dtp, "-D__SVR4") == NULL ||
976      dt_cpp_add_arg(dtp, "-D__SUNW_D=1") == NULL ||
977      dt_cpp_add_arg(dtp, isadef) == NULL ||
978      dt_cpp_add_arg(dtp, utsdef) == NULL)
979      return (set_open_errno(dtp, errp, EDT_NOMEM));

981  if (flags & DTRACE_O_NODEV)
982      bcopy(&dtrace_conf, &dtp->dt_conf, sizeof (_dtrace_conf));
983  else if (dt_ioctl(dtp, DTRACEIOC_CONF, &dtp->dt_conf) != 0)
984      return (set_open_errno(dtp, errp, errno));

986  if (flags & DTRACE_O_LP64)
987      dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_LP64;
988  else if (flags & DTRACE_O_ILP32)
989      dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_ILP32;

991  #ifdef __sparc
992  /*
993   * On SPARC systems, __sparc is always defined for <sys/isa_defs.h>
994   * and __sparcv9 is defined if we are doing a 64-bit compile.
995   */
996  if (dt_cpp_add_arg(dtp, "-D__sparc") == NULL)

```

```

997         return (set_open_errno(dtp, errp, EDT_NOMEM));

999     if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64 &&
1000         dt_cpp_add_arg(dtp, "-D__sparcv9") == NULL)
1001         return (set_open_errno(dtp, errp, EDT_NOMEM));
1002 #endif

1004 #ifdef __x86
1005 /*
1006  * On x86 systems, __i386 is defined for <sys/isa_defs.h> for 32-bit
1007  * compiles and __amd64 is defined for 64-bit compiles. Unlike SPARC,
1008  * they are defined exclusive of one another (see PSARC 2004/619).
1009  */
1010 if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64) {
1011     if (dt_cpp_add_arg(dtp, "-D__amd64") == NULL)
1012         return (set_open_errno(dtp, errp, EDT_NOMEM));
1013 } else {
1014     if (dt_cpp_add_arg(dtp, "-D__i386") == NULL)
1015         return (set_open_errno(dtp, errp, EDT_NOMEM));
1016 }
1017 #endif

1019 if (dtp->dt_conf.dtc_difversion < DIF_VERSION)
1020     return (set_open_errno(dtp, errp, EDT_DIFVERS));

1022 if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32)
1023     bcopy(_dtrace_ints_32, dtp->dt_ints, sizeof (_dtrace_ints_32));
1024 else
1025     bcopy(_dtrace_ints_64, dtp->dt_ints, sizeof (_dtrace_ints_64));

1027 dtp->dt_macros = dt_idhash_create("macro", NULL, 0, UINT_MAX);
1028 dtp->dt_aggs = dt_idhash_create("aggregation", NULL,
1029     DTRACE_AGGVARIDNONE + 1, UINT_MAX);

1031 dtp->dt_globals = dt_idhash_create("global", _dtrace_globals,
1032     DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1034 dtp->dt_tls = dt_idhash_create("thread local", NULL,
1035     DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1037 if (dtp->dt_macros == NULL || dtp->dt_aggs == NULL ||
1038     dtp->dt_globals == NULL || dtp->dt_tls == NULL)
1039     return (set_open_errno(dtp, errp, EDT_NOMEM));

1041 /*
1042  * Populate the dt_macros identifier hash table by hand: we can't use
1043  * the dt_idhash_populate() mechanism because we're not yet compiling
1044  * and dtrace_update() needs to immediately reference these ids.
1045  */
1046 for (idp = _dtrace_macros; idp->di_name != NULL; idp++) {
1047     if (dt_idhash_insert(dtp->dt_macros, idp->di_name,
1048         idp->di_kind, idp->di_flags, idp->di_id, idp->di_attr,
1049         idp->di_vers, idp->di_ops ? idp->di_ops : &dt_idops_thaw,
1050         idp->di_iarg, 0) == NULL)
1051         return (set_open_errno(dtp, errp, EDT_NOMEM));
1052 }

1054 /*
1055  * Update the module list using /system/object and load the values for
1056  * the macro variable definitions according to the current process.
1057  */
1058 dtrace_update(dtp);

1060 /*
1061  * Select the intrinsics and typedefs we want based on the data model.
1062  * The intrinsics are under "C". The typedefs are added under "D".

```

```

1063 */
1064 if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32) {
1065     dinp = _dtrace_intrinsics_32;
1066     dtyp = _dtrace_typedefs_32;
1067 } else {
1068     dinp = _dtrace_intrinsics_64;
1069     dtyp = _dtrace_typedefs_64;
1070 }

1072 /*
1073  * Create a dynamic CTF container under the "C" scope for intrinsic
1074  * types and types defined in ANSI-C header files that are included.
1075  */
1076 if ((dmp = dtp->dt_cdefs = dt_module_create(dtp, "C")) == NULL)
1077     return (set_open_errno(dtp, errp, EDT_NOMEM));

1079 if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1080     return (set_open_errno(dtp, errp, EDT_CTF));

1082 dt_dprintf("created CTF container for %s (%p)\n",
1083     dmp->dm_name, (void *)dmp->dm_ctfp);

1085 (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1086 ctf_setspecific(dmp->dm_ctfp, dmp);

1088 dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1089 dmp->dm_modid = -1; /* no module ID */

1091 /*
1092  * Fill the dynamic "C" CTF container with all of the intrinsic
1093  * integer and floating-point types appropriate for this data model.
1094  */
1095 for (; dinp->din_name != NULL; dinp++) {
1096     if (dinp->din_kind == CTF_K_INTEGER) {
1097         err = ctf_add_integer(dmp->dm_ctfp, CTF_ADD_ROOT,
1098             dinp->din_name, &dinp->din_data);
1099     } else {
1100         err = ctf_add_float(dmp->dm_ctfp, CTF_ADD_ROOT,
1101             dinp->din_name, &dinp->din_data);
1102     }

1104     if (err == CTF_ERR) {
1105         dt_dprintf("failed to add %s to C container: %s\n",
1106             dinp->din_name, ctf_errmsg(
1107                 ctf_errno(dmp->dm_ctfp)));
1108         return (set_open_errno(dtp, errp, EDT_CTF));
1109     }
1110 }

1112 if (ctf_update(dmp->dm_ctfp) != 0) {
1113     dt_dprintf("failed to update C container: %s\n",
1114         ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1115     return (set_open_errno(dtp, errp, EDT_CTF));
1116 }

1118 /*
1119  * Add intrinsic pointer types that are needed to initialize printf
1120  * format dictionary types (see table in dt_printf.c).
1121  */
1122 (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1123     ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1125 (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1126     ctf_lookup_by_name(dmp->dm_ctfp, "char"));

1128 (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,

```

```

1129     ctf_lookup_by_name(dmp->dm_ctfp, "int"));
1131     if (ctf_update(dmp->dm_ctfp) != 0) {
1132         dt_dprintf("failed to update C container: %s\n",
1133             ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1134         return (set_open_errno(dtp, errp, EDT_CTF));
1135     }
1137     /*
1138     * Create a dynamic CTF container under the "D" scope for types that
1139     * are defined by the D program itself or on-the-fly by the D compiler.
1140     * The "D" CTF container is a child of the "C" CTF container.
1141     */
1142     if ((dmp = dtp->dt_ddefs = dt_module_create(dtp, "D")) == NULL)
1143         return (set_open_errno(dtp, errp, EDT_NOMEM));
1145     if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1146         return (set_open_errno(dtp, errp, EDT_CTF));
1148     dt_dprintf("created CTF container for %s (%p)\n",
1149         dmp->dm_name, (void *)dmp->dm_ctfp);
1151     (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1152     ctf_setspecific(dmp->dm_ctfp, dmp);
1154     dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1155     dmp->dm_modid = -1; /* no module ID */
1157     if (ctf_import(dmp->dm_ctfp, dtp->dt_cdefs->dm_ctfp) == CTF_ERR) {
1158         dt_dprintf("failed to import D parent container: %s\n",
1159             ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1160         return (set_open_errno(dtp, errp, EDT_CTF));
1161     }
1163     /*
1164     * Fill the dynamic "D" CTF container with all of the built-in typedefs
1165     * that we need to use for our D variable and function definitions.
1166     * This ensures that basic inttypes.h names are always available to us.
1167     */
1168     for (; dtyp->dt_src != NULL; dtyp++) {
1169         if (ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1170             dtyp->dt_dst, ctf_lookup_by_name(dmp->dm_ctfp,
1171                 dtyp->dt_src)) == CTF_ERR) {
1172             dt_dprintf("failed to add typedef %s %s to D "
1173                 "container: %s", dtyp->dt_src, dtyp->dt_dst,
1174                 ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1175             return (set_open_errno(dtp, errp, EDT_CTF));
1176         }
1177     }
1179     /*
1180     * Insert a CTF ID corresponding to a pointer to a type of kind
1181     * CTF_K_FUNCTION we can use in the compiler for function pointers.
1182     * CTF treats all function pointers as "int (*)()" so we only need one.
1183     */
1184     ctc.ctc_return = ctf_lookup_by_name(dmp->dm_ctfp, "int");
1185     ctc.ctc_argc = 0;
1186     ctc.ctc_flags = 0;
1188     dtp->dt_type_func = ctf_add_function(dmp->dm_ctfp,
1189         CTF_ADD_ROOT, &ctc, NULL);
1191     dtp->dt_type_fptr = ctf_add_pointer(dmp->dm_ctfp,
1192         CTF_ADD_ROOT, dtp->dt_type_func);
1194     /*

```

```

1195     * We also insert CTF definitions for the special D intrinsic types
1196     * string and <DYN> into the D container. The string type is added
1197     * as a typedef of char[n]. The <DYN> type is an alias for void.
1198     * We compare types to these special CTF ids throughout the compiler.
1199     */
1200     ctr.ctr_contents = ctf_lookup_by_name(dmp->dm_ctfp, "char");
1201     ctr.ctr_index = ctf_lookup_by_name(dmp->dm_ctfp, "long");
1202     ctr.ctr_nelems = _dtrace_strsize;
1204     dtp->dt_type_str = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1205         "string", ctf_add_array(dmp->dm_ctfp, CTF_ADD_ROOT, &ctr));
1207     dtp->dt_type_dyn = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1208         "<DYN>", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1210     dtp->dt_type_stack = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1211         "stack", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1213     dtp->dt_type_symaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1214         "_symaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1216     dtp->dt_type_usymaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1217         "_usymaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));
1219     if (dtp->dt_type_func == CTF_ERR || dtp->dt_type_fptr == CTF_ERR ||
1220         dtp->dt_type_str == CTF_ERR || dtp->dt_type_dyn == CTF_ERR ||
1221         dtp->dt_type_stack == CTF_ERR || dtp->dt_type_symaddr == CTF_ERR ||
1222         dtp->dt_type_usymaddr == CTF_ERR) {
1223         dt_dprintf("failed to add intrinsic to D container: %s\n",
1224             ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1225         return (set_open_errno(dtp, errp, EDT_CTF));
1226     }
1228     if (ctf_update(dmp->dm_ctfp) != 0) {
1229         dt_dprintf("failed update D container: %s\n",
1230             ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1231         return (set_open_errno(dtp, errp, EDT_CTF));
1232     }
1234     /*
1235     * Initialize the integer description table used to convert integer
1236     * constants to the appropriate types. Refer to the comments above
1237     * dt_node_int() for a complete description of how this table is used.
1238     */
1239     for (i = 0; i < sizeof (dtp->dt_ints) / sizeof (dtp->dt_ints[0]); i++) {
1240         if (dtrace_lookup_by_type(dtp, DTRACE_OBJ_EVERY,
1241             dtp->dt_ints[i].did_name, &dti) != 0) {
1242             dt_dprintf("failed to lookup integer type %s: %s\n",
1243                 dtp->dt_ints[i].did_name,
1244                 dtrace_errmsg(dtp, dtrace_errno(dtp)));
1245             return (set_open_errno(dtp, errp, dtp->dt_errno));
1246         }
1247         dtp->dt_ints[i].did_ctfp = dti.dti_ctfp;
1248         dtp->dt_ints[i].did_type = dti.dti_type;
1249     }
1251     /*
1252     * Now that we've created the "C" and "D" containers, move them to the
1253     * start of the module list so that these types and symbols are found
1254     * first (for stability) when iterating through the module list.
1255     */
1256     dt_list_delete(&dtp->dt_modlist, dtp->dt_ddefs);
1257     dt_list_prepend(&dtp->dt_modlist, dtp->dt_ddefs);
1259     dt_list_delete(&dtp->dt_modlist, dtp->dt_cdefs);
1260     dt_list_prepend(&dtp->dt_modlist, dtp->dt_cdefs);

```



```

1262     if (dt_pfdict_create(dtp) == -1)
1263         return (set_open_errno(dtp, errp, dtp->dt_errno));

1265     /*
1266      * If we are opening libdtrace DTRACE_O_NODEV enable C_ZDEFS by default
1267      * because without /dev/dtrace open, we will not be able to load the
1268      * names and attributes of any providers or probes from the kernel.
1269      */
1270     if (flags & DTRACE_O_NODEV)
1271         dtp->dt_cflags |= DTRACE_C_ZDEFS;

1273     /*
1274      * Load hard-wired inlines into the definition cache by calling the
1275      * compiler on the raw definition string defined above.
1276      */
1277     if ((pgp = dtrace_program_strcompile(dtp, _dtrace_hardwire,
1278         DTRACE_PROBESPEC_NONE, DTRACE_C_EMPTY, 0, NULL)) == NULL) {
1279         dt_dprintf("failed to load hard-wired definitions: %s\n",
1280             dtrace_errmsg(dtp, dtrace_errno(dtp)));
1281         return (set_open_errno(dtp, errp, EDT_HARDWIRE));
1282     }

1284     dt_program_destroy(dtp, pgp);

1286     /*
1287      * Set up the default DTrace library path. Once set, the next call to
1288      * dt_compile() will compile all the libraries. We intentionally defer
1289      * library processing to improve overhead for clients that don't ever
1290      * compile, and to provide better error reporting (because the full
1291      * reporting of compiler errors requires dtrace_open() to succeed).
1292      */
1293     if (dtrace_setopt(dtp, "libdir", _dtrace_libdir) != 0)
1294         return (set_open_errno(dtp, errp, dtp->dt_errno));

1296     return (dtp);
1297 }

1299 dtrace_hdl_t *
1300 dtrace_open(int version, int flags, int *errp)
1301 {
1302     return (dt_vopen(version, flags, errp, NULL, NULL));
1303 }

1305 dtrace_hdl_t *
1306 dtrace_vopen(int version, int flags, int *errp,
1307     const dtrace_vector_t *vector, void *arg)
1308 {
1309     return (dt_vopen(version, flags, errp, vector, arg));
1310 }

1312 void
1313 dtrace_close(dtrace_hdl_t *dtp)
1314 {
1315     dt_ident_t *idp, *ndp;
1316     dt_module_t *dmp;
1317     dt_provider_t *pvp;
1318     dtrace_prog_t *pgp;
1319     dt_xlator_t *dxdp;
1320     dt_dirpath_t *dirp;
1321     int i;

1323     if (dtp->dt_procs != NULL)
1324         dt_proc_hash_destroy(dtp);

1326     while ((pgp = dt_list_next(&dtp->dt_programs)) != NULL)

```

```

1327         dt_program_destroy(dtp, pgp);

1329     while ((dxdp = dt_list_next(&dtp->dt_xlators)) != NULL)
1330         dt_xlator_destroy(dtp, dxdp);

1332     dt_free(dtp, dtp->dt_xlatormap);

1334     for (idp = dtp->dt_externs; idp != NULL; idp = ndp) {
1335         ndp = idp->di_next;
1336         dt_ident_destroy(idp);
1337     }

1339     if (dtp->dt_macros != NULL)
1340         dt_idhash_destroy(dtp->dt_macros);
1341     if (dtp->dt_aggs != NULL)
1342         dt_idhash_destroy(dtp->dt_aggs);
1343     if (dtp->dt_globals != NULL)
1344         dt_idhash_destroy(dtp->dt_globals);
1345     if (dtp->dt_tls != NULL)
1346         dt_idhash_destroy(dtp->dt_tls);

1348     while ((dmp = dt_list_next(&dtp->dt_modlist)) != NULL)
1349         dt_module_destroy(dtp, dmp);

1351     while ((pvp = dt_list_next(&dtp->dt_provlist)) != NULL)
1352         dt_provider_destroy(dtp, pvp);

1354     if (dtp->dt_fd != -1)
1355         (void) close(dtp->dt_fd);
1356     if (dtp->dt_ftfd != -1)
1357         (void) close(dtp->dt_ftfd);
1358     if (dtp->dt_cdefs_fd != -1)
1359         (void) close(dtp->dt_cdefs_fd);
1360     if (dtp->dt_ddefs_fd != -1)
1361         (void) close(dtp->dt_ddefs_fd);
1362     if (dtp->dt_stdout_fd != -1)
1363         (void) close(dtp->dt_stdout_fd);

1365     dt_epid_destroy(dtp);
1366     dt_aggid_destroy(dtp);
1367     dt_format_destroy(dtp);
1368     dt_strdata_destroy(dtp);
1369     dt_buffered_destroy(dtp);
1370     dt_aggregate_destroy(dtp);
1371     free(dtp->dt_buf.dtbd_data);
1372     dt_pfdict_destroy(dtp);
1373     dt_provmod_destroy(&dtp->dt_provmod);
1374     dt_dof_fini(dtp);

1376     for (i = 1; i < dtp->dt_cpp_argc; i++)
1377         free(dtp->dt_cpp_argv[i]);

1379     while ((dirp = dt_list_next(&dtp->dt_lib_path)) != NULL) {
1380         dt_list_delete(&dtp->dt_lib_path, dirp);
1381         free(dirp->dir_path);
1382         free(dirp);
1383     }

1385     free(dtp->dt_cpp_argv);
1386     free(dtp->dt_cpp_path);
1387     free(dtp->dt_ld_path);

1389     free(dtp->dt_mods);
1390     free(dtp->dt_provs);
1391     free(dtp);
1392 }

```

```
1394 int
1395 dtrace_provider_modules(dtrace_hdl_t *dtp, const char **mods, int nmods)
1396 {
1397     dt_provmod_t *prov;
1398     int i = 0;
1399
1400     for (prov = dtp->dt_provmod; prov != NULL; prov = prov->dp_next, i++) {
1401         if (i < nmods)
1402             mods[i] = prov->dp_name;
1403     }
1404
1405     return (i);
1406 }
1407
1408 int
1409 dtrace_ctlfd(dtrace_hdl_t *dtp)
1410 {
1411     return (dtp->dt_fd);
1412 }
```

new/usr/src/lib/libdtrace/common/ip.d.in

1

```
*****
14572 Sat Aug 18 10:37:03 2012
new/usr/src/lib/libdtrace/common/ip.d.in
dccp: sock upcalls
*****

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 */

25 #pragma D depends_on module ip
26 #pragma D depends_on provider ip

28 inline int IPH_DF = @IPH_DF@;
29 #pragma D binding "1.5" IPH_DF
30 inline int IPH_MF = @IPH_MF@;
31 #pragma D binding "1.5" IPH_MF

33 #pragma D binding "1.5" IPPROTO_IP
34 inline int IPPROTO_IP = @IPPROTO_IP@;
35 #pragma D binding "1.5" IPPROTO_HOPOPTS
36 inline int IPPROTO_HOPOPTS = @IPPROTO_HOPOPTS@;
37 #pragma D binding "1.5" IPPROTO_ICMP
38 inline int IPPROTO_ICMP = @IPPROTO_ICMP@;
39 #pragma D binding "1.5" IPPROTO_IGMP
40 inline int IPPROTO_IGMP = @IPPROTO_IGMP@;
41 #pragma D binding "1.5" IPPROTO_GGP
42 inline int IPPROTO_GGP = @IPPROTO_GGP@;
43 #pragma D binding "1.5" IPPROTO_ENCAP
44 inline int IPPROTO_ENCAP = @IPPROTO_ENCAP@;
45 #pragma D binding "1.5" IPPROTO_TCP
46 inline int IPPROTO_TCP = @IPPROTO_TCP@;
47 #pragma D binding "1.5" IPPROTO_EGP
48 inline int IPPROTO_EGP = @IPPROTO_EGP@;
49 #pragma D binding "1.5" IPPROTO_PUP
50 inline int IPPROTO_PUP = @IPPROTO_PUP@;
51 #pragma D binding "1.5" IPPROTO_UDP
52 inline int IPPROTO_UDP = @IPPROTO_UDP@;
53 #pragma D binding "1.5" IPPROTO_IDP
54 inline int IPPROTO_IDP = @IPPROTO_IDP@;
55 #pragma D binding "1.5" IPPROTO_IPV6
56 inline int IPPROTO_IPV6 = @IPPROTO_IPV6@;
57 #pragma D binding "1.5" IPPROTO_ROUTING
58 inline int IPPROTO_ROUTING = @IPPROTO_ROUTING@;
59 #pragma D binding "1.5" IPPROTO_FRAGMENT
60 inline int IPPROTO_FRAGMENT = @IPPROTO_FRAGMENT@;
61 #pragma D binding "1.5" IPPROTO_RSVP
```

new/usr/src/lib/libdtrace/common/ip.d.in

2

```
62 inline int IPPROTO_RSVP = @IPPROTO_RSVP@;
63 #pragma D binding "1.5" IPPROTO_ESP
64 inline int IPPROTO_ESP = @IPPROTO_ESP@;
65 #pragma D binding "1.5" IPPROTO_AH
66 inline int IPPROTO_AH = @IPPROTO_AH@;
67 #pragma D binding "1.5" IPPROTO_ICMPV6
68 inline int IPPROTO_ICMPV6 = @IPPROTO_ICMPV6@;
69 #pragma D binding "1.5" IPPROTO_NONE
70 inline int IPPROTO_NONE = @IPPROTO_NONE@;
71 #pragma D binding "1.5" IPPROTO_DSTOPTS
72 inline int IPPROTO_DSTOPTS = @IPPROTO_DSTOPTS@;
73 #pragma D binding "1.5" IPPROTO_HELLO
74 inline int IPPROTO_HELLO = @IPPROTO_HELLO@;
75 #pragma D binding "1.5" IPPROTO_ND
76 inline int IPPROTO_ND = @IPPROTO_ND@;
77 #pragma D binding "1.5" IPPROTO_EON
78 inline int IPPROTO_EON = @IPPROTO_EON@;
79 #pragma D binding "1.5" IPPROTO_OSPF
80 inline int IPPROTO_OSPF = @IPPROTO_OSPF@;
81 #pragma D binding "1.5" IPPROTO_PIM
82 inline int IPPROTO_PIM = @IPPROTO_PIM@;
83 #pragma D binding "1.5" IPPROTO_SCTP
84 inline int IPPROTO_SCTP = @IPPROTO_SCTP@;
85 #pragma D binding "1.5" IPPROTO_RAW
86 inline int IPPROTO_RAW = @IPPROTO_RAW@;
87 #pragma D binding "1.5" IPPROTO_MAX
88 inline int IPPROTO_MAX = @IPPROTO_MAX@;
89 #pragma D binding "1.10" IPPROTO_DCCP
90 inline int IPPROTO_DCCP = @IPPROTO_DCCP@;
91 #endif /* ! codereview */

93 /*
94  * pktinfo is where packet ID info can be made available for deeper
95  * analysis if packet IDs become supported by the kernel in the future.
96  * The pkt_addr member is currently always NULL.
97  */
98 typedef struct pktinfo {
99     uintptr_t pkt_addr;
100 } pktinfo_t;

102 /*
103  * csinfo is where connection state info is made available.
104  */
105 typedef struct csinfo {
106     uintptr_t cs_addr;
107     uint64_t cs_cid;
108     pid_t cs_pid;
109     zoneid_t cs_zoneid;
110 } csinfo_t;

112 /*
113  * ipinfo contains common IP info for both IPv4 and IPv6.
114  */
115 typedef struct ipinfo {
116     uint8_t ip_ver; /* IP version (4, 6) */
117     uint32_t ip_plength; /* payload length */
118     string ip_saddr; /* source address */
119     string ip_daddr; /* destination address */
120 } ipinfo_t;

122 /*
123  * ifinfo contains network interface info.
124  */
125 typedef struct ifinfo {
126     string if_name; /* interface name */
127     int8_t if_local; /* is delivered locally */
128 }
```

```

128     netstackid_t if_ipstack;      /* ipstack ID */
129     uintptr_t if_addr;           /* pointer to raw ill_t */
130 } ifinfo_t;

132 /*
133  * ipv4info is a translated version of the IPv4 header (with raw pointer).
134  * These values are NULL if the packet is not IPv4.
135  */
136 typedef struct ipv4info {
137     uint8_t ipv4_ver;             /* IP version (4) */
138     uint8_t ipv4_ihl;             /* header length, bytes */
139     uint8_t ipv4_tos;             /* type of service field */
140     uint16_t ipv4_length;         /* length (header + payload) */
141     uint16_t ipv4_ident;          /* identification */
142     uint8_t ipv4_flags;           /* IP flags */
143     uint16_t ipv4_offset;         /* fragment offset */
144     uint8_t ipv4_ttl;             /* time to live */
145     uint8_t ipv4_protocol;        /* next level protocol */
146     string ipv4_protostr;         /* next level protocol, as a string */
147     uint16_t ipv4_checksum;       /* header checksum */
148     ipaddr_t ipv4_src;            /* source address */
149     ipaddr_t ipv4_dst;            /* destination address */
150     string ipv4_saddr;            /* source address, string */
151     string ipv4_daddr;            /* destination address, string */
152     ipha_t *ipv4_hdr;             /* pointer to raw header */
153 } ipv4info_t;

155 /*
156  * ipv6info is a translated version of the IPv6 header (with raw pointer).
157  * These values are NULL if the packet is not IPv6.
158  */
159 typedef struct ipv6info {
160     uint8_t ipv6_ver;             /* IP version (6) */
161     uint8_t ipv6_tclass;          /* traffic class */
162     uint32_t ipv6_flow;           /* flow label */
163     uint16_t ipv6_plen;           /* payload length */
164     uint8_t ipv6_nextthr;         /* next header protocol */
165     string ipv6_nextstr;          /* next header protocol, as a string */
166     uint8_t ipv6_hlim;            /* hop limit */
167     in6_addr_t *ipv6_src;         /* source address */
168     in6_addr_t *ipv6_dst;         /* destination address */
169     string ipv6_saddr;            /* source address, string */
170     string ipv6_daddr;            /* destination address, string */
171     ip6_t *ipv6_hdr;             /* pointer to raw header */
172 } ipv6info_t;

174 /*
175  * void_ip_t is a void pointer to either an IPv4 or IPv6 header. It has
176  * its own type name so that a translator can be determined.
177  */
178 typedef uintptr_t void_ip_t;

180 /*
181  * __dtrace_ipsr_ill_t is used by the translator to take an ill_t plus an
182  * additional arg6 from the ip:::send and ip:::recieve probes, and translate
183  * them to an ifinfo_t.
184  */
185 typedef ill_t __dtrace_ipsr_ill_t;

187 /*
188  * __dtrace_tcp_void_ip_t is used by the translator to take either the
189  * non-NULL void_ip_t * passed in or, if it is NULL, uses arg3 (tcp_t *)
190  * from the tcp:::send and tcp:::recieve probes to translate to an ipinfo_t.
191  * When no headers are available in the TCP fusion case for tcp:::send
192  * and tcp:::receive case, this allows us to present the consumer with header
193  * data based on the tcp_t * content in order to hide the implementation

```

```

194  * details of TCP fusion.
195  */
196 typedef void * __dtrace_tcp_void_ip_t;

198 #pragma D binding "1.5" translator
199 translator pktinfo_t < mblk_t *M > {
200     pkt_addr = NULL;
201 };

203 #pragma D binding "1.5" translator
204 translator csinfo_t < conn_t *C > {
205     cs_addr = NULL;
206 };

208 #pragma D binding "1.6.3" translator
209 translator csinfo_t < ip_xmit_attr_t *C > {
210     cs_addr = (uintptr_t)C;
211     cs_cid = C ? C->ixa_conn_id : NULL;
212     cs_pid = C ? C->ixa_cpuid : -1;
213     cs_zoneid = C ?
214         (C->ixa_ipst == NULL || C->ixa_ipst->ips_netstack == NULL ||
215          C->ixa_ipst->ips_netstack->netstack_stackid ==
216           @GLOBAL_NETSTACKID@ ||
217          C->ixa_cred == NULL ||
218          C->ixa_cred->cr_zone == NULL ||
219          C->ixa_cred->cr_uid == -1 ?
220          C->ixa_zoneid : C->ixa_cred->cr_zone->zone_id) : -1;
221 };

223 #pragma D binding "1.5" translator
224 translator ipinfo_t < ipha_t *I > {
225     ip_ver = I->ipha_version_and_hdr_length >> 4;
226     ip_plength = ntohs(I->ipha_length) -
227         ((I->ipha_version_and_hdr_length & 0xf) << 2);
228     ip_saddr = inet_ntoa(&I->ipha_src);
229     ip_daddr = inet_ntoa(&I->ipha_dst);
230 };

232 #pragma D binding "1.5" translator
233 translator ipinfo_t < ip6_t *I > {
234     ip_ver = *(uint8_t *)I >> 4;
235     ip_plength = ntohs(I->ip6_ctlun.ip6_un1.ip6_un1_plen);
236     ip_saddr = inet_ntoa6(&I->ip6_src);
237     ip_daddr = inet_ntoa6(&I->ip6_dst);
238 };

240 #pragma D binding "1.5" translator
241 translator ipinfo_t < void_ip_t *I > {
242     ip_ver = I != NULL ? *(uint8_t *)I >> 4 : 0;
243     ip_plength = I != NULL ? (*(uint8_t *)I >> 4 == 4 ?
244         ntohs(((ipha_t *)I)->ipha_length) -
245         (((ipha_t *)I)->ipha_version_and_hdr_length & 0xf) << 2) :
246         *(uint8_t *)I >> 4 == 6 ?
247         ntohs(((ip6_t *)I)->ip6_ctlun.ip6_un1.ip6_un1_plen) : 0) : 0;
248     ip_saddr = I != NULL ? (*(uint8_t *)I >> 4 == 4 ?
249         inet_ntoa(&((ipha_t *)I)->ipha_src) : *(uint8_t *)I >> 4 == 6 ?
250         inet_ntoa6(&((ip6_t *)I)->ip6_src) : "<unknown>") : "<unknown>";
251     ip_daddr = I != NULL ? (*(uint8_t *)I >> 4 == 4 ?
252         inet_ntoa(&((ipha_t *)I)->ipha_dst) : *(uint8_t *)I >> 4 == 6 ?
253         inet_ntoa6(&((ip6_t *)I)->ip6_dst) : "<unknown>") : "<unknown>";
254 };

256 #pragma D binding "1.5" translator
257 translator ipinfo_t < __dtrace_ipsr_ill_t *I > {
258     if_name = I != NULL ? stringof(I->ill_name) : "<null>";
259     if_ipstack = I != NULL ? I->ill_ipst->ips_netstack->netstack_stackid

```

```

260     : 0;
261     if_local = arg6;                /* probe dependent */
262     if_addr = (uintptr_t)I;
263 };

265 /*
266  * Translate to an ipinfo_t * from either the non-NULL void_ip_t * passed in,
267  * or use arg3 (tcp_t *) to fabricate ip header info.
268  */
269 #pragma D binding "1.6.3" translator
270 translator ipinfo_t < __dtrace_tcp_void_ip_t *I > {
271     ip_ver = I != NULL ? *(uint8_t *)I >> 4 :
272     arg3 != NULL ? ((tcp_t *)arg3)->tcp_connp->conn_ipversion : 0;
273     ip_plen =
274     I != NULL && *(uint8_t *)I >> 4 == 4 ?
275     ntohs(((ipha_t *)I)->ipha_length) -
276     (((ipha_t *)I)->ipha_version_and_hdr_length & 0xf) << 2 :
277     I != NULL && *(uint8_t *)I >> 4 == 6 ?
278     ntohs(((ip6_t *)I)->ip6_ctlun.ip6_un1.ip6_un1_plen) :
279     I != NULL ? 0 :
280     arg3 != NULL && probename == "send" ?
281     ((tcp_t *)arg3)->tcp_last_sent_len + @TCP_MIN_HEADER_LENGTH :
282     arg3 != NULL && probename == "receive" ?
283     ((tcp_t *)arg3)->tcp_last_recv_len + @TCP_MIN_HEADER_LENGTH :
284     0;
285     ip_saddr =
286     I != NULL && *(uint8_t *)I >> 4 == 4 ?
287     inet_ntoa(&((ipha_t *)I)->ipha_src) :
288     I != NULL && *(uint8_t *)I >> 4 == 6 ?
289     inet_ntoa6(&((ip6_t *)I)->ip6_src) :
290     I != NULL ? "<unknown>" :
291     arg3 != NULL && probename == "send" ?
292     inet_ntoa6(&((tcp_t *)arg3)->tcp_connp->connua_v6addr.connua_laddr) :
293     arg3 != NULL && probename == "receive" ?
294     inet_ntoa6(&((tcp_t *)arg3)->tcp_connp->connua_v6addr.connua_faddr) :
295     "<unknown>";
296     ip_daddr =
297     I != NULL && *(uint8_t *)I >> 4 == 4 ?
298     inet_ntoa(&((ipha_t *)I)->ipha_dst) :
299     I != NULL && *(uint8_t *)I >> 4 == 6 ?
300     inet_ntoa6(&((ip6_t *)I)->ip6_dst) :
301     I != NULL ? "<unknown>" :
302     arg3 != NULL && probename == "send" ?
303     inet_ntoa6(&((tcp_t *)arg3)->tcp_connp->connua_v6addr.connua_faddr) :
304     arg3 != NULL && probename == "receive" ?
305     inet_ntoa6(&((tcp_t *)arg3)->tcp_connp->connua_v6addr.connua_laddr) :
306     "<unknown>";
307 };

309 #pragma D binding "1.5" translator
310 translator ipv4info_t < ipha_t *I > {
311     ipv4_ver = I != NULL ? I->ipha_version_and_hdr_length >> 4 : 0;
312     ipv4_ihl = I != NULL ? (I->ipha_version_and_hdr_length & 0xf) << 2 : 0;
313     ipv4_tos = I != NULL ? I->ipha_type_of_service : 0;
314     ipv4_length = I != NULL ? ntohs(I->ipha_length) : 0;
315     ipv4_ident = I != NULL ? ntohs(I->ipha_ident) : 0;
316     ipv4_flags = I != NULL ? ntohs(I->ipha_fragment_offset_and_flags) >>
317     12 : 0;
318     ipv4_offset = I != NULL ? ntohs(I->ipha_fragment_offset_and_flags) &
319     0x0fff : 0;
320     ipv4_ttl = I != NULL ? I->ipha_ttl : 0;
321     ipv4_protocol = I != NULL ? I->ipha_protocol : 0;
322     ipv4_protostr = I == NULL ? "<null>" :
323     I->ipha_protocol == IPPROTO_TCP ? "TCP" :
324     I->ipha_protocol == IPPROTO_UDP ? "UDP" :
325     I->ipha_protocol == IPPROTO_IP ? "IP" :

```

```

326     I->ipha_protocol == IPPROTO_ICMP ? "ICMP" :
327     I->ipha_protocol == IPPROTO_IGMP ? "IGMP" :
328     I->ipha_protocol == IPPROTO_EGP ? "EGP" :
329     I->ipha_protocol == IPPROTO_IPV6 ? "IPv6" :
330     I->ipha_protocol == IPPROTO_ROUTING ? "ROUTE" :
331     I->ipha_protocol == IPPROTO_ESP ? "ESP" :
332     I->ipha_protocol == IPPROTO_AH ? "AH" :
333     I->ipha_protocol == IPPROTO_ICMPV6 ? "ICMPv6" :
334     I->ipha_protocol == IPPROTO_OSPF ? "OSPF" :
335     I->ipha_protocol == IPPROTO_SCTP ? "SCTP" :
336     I->ipha_protocol == IPPROTO_RAW ? "RAW" :
337     lltostr((uint64_t)I->ipha_protocol);
338     ipv4_checksum = I != NULL ? ntohs(I->ipha_hdr_checksum) : 0;
339     ipv4_src = I != NULL ? I->ipha_src : 0;
340     ipv4_dst = I != NULL ? I->ipha_dst : 0;
341     ipv4_saddr = I != NULL ? inet_ntoa(&I->ipha_src) : "<null>";
342     ipv4_daddr = I != NULL ? inet_ntoa(&I->ipha_dst) : "<null>";
343     ipv4_hdr = I;
344 };

346 #pragma D binding "1.5" translator
347 translator ipv6info_t < ip6_t *I > {
348     ipv6_ver = I != NULL ? I->ip6_ctlun.ip6_un2_vfc >> 4 : 0;
349     ipv6_tclass = I != NULL ? ((I->ip6_ctlun.ip6_un1.ip6_un1_flow &&
350     0x0fffffff) >> 20) : 0;
351     ipv6_flow = I != NULL ? I->ip6_ctlun.ip6_un1.ip6_un1_flow &&
352     0x000ffff : 0;
353     ipv6_plen = I != NULL ? ntohs(I->ip6_ctlun.ip6_un1.ip6_un1_plen) : 0;
354     ipv6nexthdr = I != NULL ? I->ip6_ctlun.ip6_un1.ip6_un1_nxt : 0;
355     ipv6nextstr = I == NULL ? "<null>" :
356     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_TCP ? "TCP" :
357     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_UDP ? "UDP" :
358     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_IP ? "IP" :
359     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_ICMP ? "ICMP" :
360     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_IGMP ? "IGMP" :
361     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_EGP ? "EGP" :
362     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_IPV6 ? "IPv6" :
363     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_ROUTING ? "ROUTE" :
364     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_ESP ? "ESP" :
365     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_AH ? "AH" :
366     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_ICMPV6 ? "ICMPv6" :
367     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_OSPF ? "OSPF" :
368     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_SCTP ? "SCTP" :
369     I->ip6_ctlun.ip6_un1.ip6_un1_nxt == IPPROTO_RAW ? "RAW" :
370     lltostr((uint64_t)I->ip6_ctlun.ip6_un1.ip6_un1_nxt);
371     ipv6_hlim = I != NULL ? I->ip6_ctlun.ip6_un1.ip6_un1_hlim : 0;
372     ipv6_src = I != NULL ? &I->ip6_src : 0;
373     ipv6_dst = I != NULL ? &I->ip6_dst : 0;
374     ipv6_saddr = I != NULL ? inet_ntoa6(&I->ip6_src) : "<null>";
375     ipv6_daddr = I != NULL ? inet_ntoa6(&I->ip6_dst) : "<null>";
376     ipv6_hdr = I;
377 };

```

new/usr/src/lib/libdtrace/common/ip.sed.in

1

```
*****
2337 Sat Aug 18 10:37:04 2012
new/usr/src/lib/libdtrace/common/ip.sed.in
dccp: sock upcalls
*****

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 */

24 /*
25 * This file is a sed script which is first preprocessed by cpp or cc -E to
26 * define a set of sed directives which replace #define tokens with their
27 * values. After preprocessing, the sed script is run over ip.d.in to
28 * replace the #define tokens listed below to create the finished ip.d.
29 * Refer to the rules in libdtrace/Makefile.com for more information.
30 */

31

32
33 #include <sys/netstack.h>
34 #include <sys/socket.h>
35 #include <netinet/in.h>
36 #include <inet/ip.h>
37 #include <inet/tcp.h>

38
39 #define SED_REPLACE(x) s/#x/x/g

40
41 SED_REPLACE(AF_INET)
42 SED_REPLACE(AF_INET6)

43
44 SED_REPLACE(IPH_DF)
45 SED_REPLACE(IPH_MF)

46
47 SED_REPLACE(IPPROTO_IP)
48 SED_REPLACE(IPPROTO_HOPOPTS)
49 SED_REPLACE(IPPROTO_ICMP)
50 SED_REPLACE(IPPROTO_IGMP)
51 SED_REPLACE(IPPROTO_GGP)
52 SED_REPLACE(IPPROTO_ENCAP)
53 SED_REPLACE(IPPROTO_TCP)
54 SED_REPLACE(IPPROTO_EGP)
55 SED_REPLACE(IPPROTO_PUP)
56 SED_REPLACE(IPPROTO_UDP)
57 SED_REPLACE(IPPROTO_IDP)
58 SED_REPLACE(IPPROTO_IPV6)
59 SED_REPLACE(IPPROTO_ROUTING)
60 SED_REPLACE(IPPROTO_FRAGMENT)
61 SED_REPLACE(IPPROTO_RSVP)
```

new/usr/src/lib/libdtrace/common/ip.sed.in

2

```
62 SED_REPLACE(IPPROTO_ESP)
63 SED_REPLACE(IPPROTO_AH)
64 SED_REPLACE(IPPROTO_ICMPV6)
65 SED_REPLACE(IPPROTO_NONE)
66 SED_REPLACE(IPPROTO_DSTOPTS)
67 SED_REPLACE(IPPROTO_HELLO)
68 SED_REPLACE(IPPROTO_ND)
69 SED_REPLACE(IPPROTO_EON)
70 SED_REPLACE(IPPROTO_OSPF)
71 SED_REPLACE(IPPROTO_PIM)
72 SED_REPLACE(IPPROTO_SCTP)
73 SED_REPLACE(IPPROTO_RAW)
74 SED_REPLACE(IPPROTO_DCCP)
75 #endif /* ! codereview */
76 SED_REPLACE(IPPROTO_MAX)

77
78 SED_REPLACE(TCP_MIN_HEADER_LENGTH)

79
80 SED_REPLACE(GLOBAL_NETSTACKID)
```

new/usr/src/lib/libipadm/common/ipadm_prop.c

1

```
*****
55005 Sat Aug 18 10:37:04 2012
new/usr/src/lib/libipadm/common/ipadm_prop.c
dccp: properties
*****

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
```

new/usr/src/lib/libipadm/common/ipadm_prop.c

2

```
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) strcpy(lifr.lifr_name, ifname, sizeof (lifr.lifr_name));
485     s = (proto == MOD_PROTO_IPV6 ? iph->iph_sock6 : iph->iph_sock);
487     if (strcmp(pval, IPADM_NONESTR) != 0) {
488         if ((ifindex = if_nametoindex(pval)) == 0)
489             return (ipadm_errno2status(errno));
490         lifr.lifr_index = ifindex;
491     } else {
492         if (ioctl(s, SIOCGLIFUSESRC, (caddr_t)&lifr) < 0)
493             return (ipadm_errno2status(errno));
494         lifr.lifr_index = 0;
495     }
496     if (ioctl(s, SIOCSLIFUSESRC, (caddr_t)&lifr) < 0)
497         return (ipadm_errno2status(errno));
499     return (IPADM_SUCCESS);
500 }

502 static struct hostmodel_strval {
503     char *esm_str;
504     ip_hostmodel_t esm_val;
505 } esm_arr[] = {
506     {"weak", IP_WEAK_ES},
507     {"src-priority", IP_SRC_PRI_ES},
508     {"strong", IP_STRONG_ES},
509     {"custom", IP_MAXVAL_ES}
510 };

512 static ip_hostmodel_t
513 i_ipadm_hostmodel_str2val(const char *pval)
514 {
515     int i;

517     for (i = 0; i < A_CNT(esm_arr); i++) {
518         if (esm_arr[i].esm_str != NULL &&
519             strcmp(pval, esm_arr[i].esm_str) == 0) {
520             return (esm_arr[i].esm_val);
521         }
522     }
523     return (IP_MAXVAL_ES);
524 }

526 static char *
527 i_ipadm_hostmodel_val2str(ip_hostmodel_t pval)
528 {
529     int i;

531     for (i = 0; i < A_CNT(esm_arr); i++) {
532         if (esm_arr[i].esm_val == pval)
533             return (esm_arr[i].esm_str);
534     }
535     return (NULL);
536 }

538 /* ARGSUSED */
539 static ipadm_status_t
540 i_ipadm_set_hostmodel(ipadm_handle_t iph, const void *arg,
541     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
542 {
543     ip_hostmodel_t hostmodel;
544     char val[11]; /* covers uint32_max as a string */
546     if ((flags & IPADM_OPT_DEFAULT) == 0) {

```

```

547         hostmodel = i_ipadm_hostmodel_str2val(pval);
548         if (hostmodel == IP_MAXVAL_ES)
549             return (IPADM_INVALID_ARG);
550         (void) snprintf(val, sizeof (val), "%d", hostmodel);
551         pval = val;
552     }
553     return (i_ipadm_set_prop(iph, NULL, pdp, pval, proto, flags));
554 }

556 /* ARGSUSED */
557 static ipadm_status_t
558 i_ipadm_get_hostmodel(ipadm_handle_t iph, const void *arg,
559     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
560     uint_t valtype)
561 {
562     ip_hostmodel_t hostmodel;
563     char *cp;
564     size_t nbytes;
565     ipadm_status_t status;

567     switch (valtype) {
568     case MOD_PROP_PERM:
569         nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
570         break;
571     case MOD_PROP_DEFAULT:
572         nbytes = snprintf(buf, *bufsize, "weak");
573         break;
574     case MOD_PROP_ACTIVE:
575         status = i_ipadm_get_prop(iph, arg, pdp, buf, bufsize, proto,
576             valtype);
577         if (status != IPADM_SUCCESS)
578             return (status);
579         bcopy(buf, &hostmodel, sizeof (hostmodel));
580         cp = i_ipadm_hostmodel_val2str(hostmodel);
581         nbytes = snprintf(buf, *bufsize, "%s",
582             (cp != NULL ? cp : "?"));
583         break;
584     case MOD_PROP_POSSIBLE:
585         nbytes = snprintf(buf, *bufsize, "strong,src-priority,weak");
586         break;
587     default:
588         return (IPADM_INVALID_ARG);
589     }
590     if (nbytes >= *bufsize) {
591         /* insufficient buffer space */
592         *bufsize = nbytes + 1;
593         return (IPADM_NO_BUFS);
594     }
595     return (IPADM_SUCCESS);
596 }

598 /* ARGSUSED */
599 static ipadm_status_t
600 i_ipadm_set_ifprop_flags(ipadm_handle_t iph, const void *arg,
601     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
602 {
603     ipadm_status_t status = IPADM_SUCCESS;
604     const char *ifname = arg;
605     uint64_t on_flags = 0, off_flags = 0;
606     boolean_t on = B_FALSE;
607     sa_family_t af = (proto == MOD_PROTO_IPV6 ? AF_INET6 : AF_INET);

609     /* if we are resetting, set the value to its default value */
610     if (flags & IPADM_OPT_DEFAULT) {
611         if (strcmp(pdp->ipd_name, "exchange_routes") == 0 ||
612             strcmp(pdp->ipd_name, "arp") == 0 ||

```

```

613         strcmp(pdp->ipd_name, "nud") == 0) {
614             pval = IPADM_ONSTR;
615         } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
616             pval = IPADM_OFFSTR;
617         } else {
618             return (IPADM_PROP_UNKNOWN);
619         }
620     }

622     if (strcmp(pval, IPADM_ONSTR) == 0)
623         on = B_TRUE;
624     else if (strcmp(pval, IPADM_OFFSTR) == 0)
625         on = B_FALSE;
626     else
627         return (IPADM_INVALID_ARG);

629     if (strcmp(pdp->ipd_name, "exchange_routes") == 0) {
630         if (on)
631             off_flags = IFF_NORTEXCH;
632         else
633             on_flags = IFF_NORTEXCH;
634     } else if (strcmp(pdp->ipd_name, "arp") == 0) {
635         if (on)
636             off_flags = IFF_NOARP;
637         else
638             on_flags = IFF_NOARP;
639     } else if (strcmp(pdp->ipd_name, "nud") == 0) {
640         if (on)
641             off_flags = IFF_NONUD;
642         else
643             on_flags = IFF_NONUD;
644     } else if (strcmp(pdp->ipd_name, "forwarding") == 0) {
645         if (on)
646             on_flags = IFF_ROUTER;
647         else
648             off_flags = IFF_ROUTER;
649     }

651     if (on_flags || off_flags) {
652         status = i_ipadm_set_flags(iph, ifname, af, on_flags,
653             off_flags);
654     }
655     return (status);
656 }

658 /* ARGSUSED */
659 static ipadm_status_t
660 i_ipadm_set_eprivport(ipadm_handle_t iph, const void *arg,
661     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
662 {
663     nvlist_t      *portsnvl = NULL;
664     nvpair_t      *nvp;
665     ipadm_status_t status = IPADM_SUCCESS;
666     int            err;
667     uint_t         count = 0;

669     if (flags & IPADM_OPT_DEFAULT) {
670         assert(pval == NULL);
671         return (i_ipadm_set_prop(iph, arg, pdp, pval, proto, flags));
672     }

674     if ((err = ipadm_str2nvlist(pval, &portsnvl, IPADM_NORVAL)) != 0)
675         return (ipadm_errno2status(err));

677     /* count the number of ports */
678     for (nvp = nvlist_next_nvpair(portsnvl, NULL); nvp != NULL;

```

```

679         nvp = nvlist_next_nvpair(portsnvl, nvp)) {
680             ++count;
681         }

683     if (iph->iph_flags & IPH_INIT) {
684         flags |= IPADM_OPT_APPEND;
685     } else if (count > 1) {
686         /*
687          * We allow only one port to be added, removed or
688          * assigned at a time.
689          *
690          * However on reboot, while initializing protocol
691          * properties, extra_priv_ports might have multiple
692          * values. Only in that case we allow setting multiple
693          * values.
694          */
695         nvlist_free(portsnvl);
696         return (IPADM_INVALID_ARG);
697     }

699     for (nvp = nvlist_next_nvpair(portsnvl, NULL); nvp != NULL;
700         nvp = nvlist_next_nvpair(portsnvl, nvp)) {
701         status = i_ipadm_set_prop(iph, arg, pdp, nvpair_name(nvp),
702             proto, flags);
703         if (status != IPADM_SUCCESS)
704             break;
705     }
706     nvlist_free(portsnvl);
707     return (status);
708 }

710 /* ARGSUSED */
711 static ipadm_status_t
712 i_ipadm_set_forwarding(ipadm_handle_t iph, const void *arg,
713     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
714 {
715     const char      *ifname = arg;
716     ipadm_status_t  status;

718     /*
719      * if interface name is provided, then set forwarding using the
720      * IFF_ROUTER flag
721      */
722     if (ifname != NULL) {
723         status = i_ipadm_set_ifprop_flags(iph, ifname, pdp, pval,
724             proto, flags);
725     } else {
726         char          *val = NULL;

728         /*
729          * if the caller is IPH_LEGACY, 'pval' already contains
730          * numeric values.
731          */
732         if (!(flags & IPADM_OPT_DEFAULT) &&
733             !(iph->iph_flags & IPH_LEGACY)) {
735             if (strcmp(pval, IPADM_ONSTR) == 0)
736                 val = "1";
737             else if (strcmp(pval, IPADM_OFFSTR) == 0)
738                 val = "0";
739             else
740                 return (IPADM_INVALID_ARG);
741             pval = val;
742         }

744         status = i_ipadm_set_prop(iph, ifname, pdp, pval, proto, flags);

```

```

745     }
746
747     return (status);
748 }
749
750 /* ARGSUSED */
751 static ipadm_status_t
752 i_ipadm_set_ecnsack(ipadm_handle_t iph, const void *arg,
753     ipadm_prop_desc_t *pdp, const void *pval, uint_t proto, uint_t flags)
754 {
755     uint_t    i;
756     char      val[MAXPROPVALLEN];
757
758     /* if IPH_LEGACY is set, 'pval' already contains numeric values */
759     if (!(flags & IPADM_OPT_DEFAULT) && !(iph->iph_flags & IPH_LEGACY)) {
760         for (i = 0; ecn_sack_vals[i] != NULL; i++) {
761             if (strcmp(pval, ecn_sack_vals[i]) == 0)
762                 break;
763         }
764         if (ecn_sack_vals[i] == NULL)
765             return (IPADM_INVALID_ARG);
766         (void) snprintf(val, MAXPROPVALLEN, "%d", i);
767         pval = val;
768     }
769
770     return (i_ipadm_set_prop(iph, arg, pdp, pval, proto, flags));
771 }
772
773 /* ARGSUSED */
774 ipadm_status_t
775 i_ipadm_get_ecnsack(ipadm_handle_t iph, const void *arg,
776     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
777     uint_t valtype)
778 {
779     ipadm_status_t    status = IPADM_SUCCESS;
780     uint_t            i, nbytes = 0;
781
782     switch (valtype) {
783     case MOD_PROP_POSSIBLE:
784         for (i = 0; ecn_sack_vals[i] != NULL; i++) {
785             if (i == 0)
786                 nbytes += snprintf(buf + nbytes,
787                     *bufsize - nbytes, "%s", ecn_sack_vals[i]);
788             else
789                 nbytes += snprintf(buf + nbytes,
790                     *bufsize - nbytes, ",%s", ecn_sack_vals[i]);
791             if (nbytes >= *bufsize)
792                 break;
793         }
794         break;
795     case MOD_PROP_PERM:
796     case MOD_PROP_DEFAULT:
797     case MOD_PROP_ACTIVE:
798         status = i_ipadm_get_prop(iph, arg, pdp, buf, bufsize, proto,
799             valtype);
800
801     /*
802      * If IPH_LEGACY is set, do not convert the value returned
803      * from kernel,
804      */
805     if (iph->iph_flags & IPH_LEGACY)
806         break;
807
808     /*
809      * For current and default value, convert the value returned
810      * from kernel to more discrete representation.

```

```

811     */
812     if (status == IPADM_SUCCESS && (valtype == MOD_PROP_ACTIVE ||
813         valtype == MOD_PROP_DEFAULT)) {
814         i = atoi(buf);
815         assert(i < 3);
816         nbytes = snprintf(buf, *bufsize, "%s",
817             ecn_sack_vals[i]);
818     }
819     break;
820 default:
821     return (IPADM_INVALID_ARG);
822 }
823 if (nbytes >= *bufsize) {
824     /* insufficient buffer space */
825     *bufsize = nbytes + 1;
826     return (IPADM_NO_BUFS);
827 }
828
829 return (status);
830 }
831
832 /* ARGSUSED */
833 static ipadm_status_t
834 i_ipadm_get_forwarding(ipadm_handle_t iph, const void *arg,
835     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
836     uint_t valtype)
837 {
838     const char    *ifname = arg;
839     ipadm_status_t    status = IPADM_SUCCESS;
840
841     /*
842      * if interface name is provided, then get forwarding status using
843      * SIOCGLIFFLAGS
844      */
845     if (ifname != NULL) {
846         status = i_ipadm_get_ifprop_flags(iph, ifname, pdp,
847             buf, bufsize, pdp->ipd_proto, valtype);
848     } else {
849         status = i_ipadm_get_prop(iph, ifname, pdp, buf,
850             bufsize, proto, valtype);
851     }
852     /*
853      * If IPH_LEGACY is set, do not convert the value returned
854      * from kernel,
855      */
856     if (iph->iph_flags & IPH_LEGACY)
857         goto ret;
858     if (status == IPADM_SUCCESS && (valtype == MOD_PROP_ACTIVE ||
859         valtype == MOD_PROP_DEFAULT)) {
860         uint_t    val = atoi(buf);
861
862         (void) snprintf(buf, *bufsize,
863             (val == 1 ? IPADM_ONSTR : IPADM_OFFSTR));
864     }
865
866 ret:
867     return (status);
868 }
869
870 /* ARGSUSED */
871 static ipadm_status_t
872 i_ipadm_get_mtu(ipadm_handle_t iph, const void *arg,
873     ipadm_prop_desc_t *pdp, char *buf, uint_t *bufsize, uint_t proto,
874     uint_t valtype)
875 {
876     struct lifreq    lifr;

```

```

877     const char    *ifname = arg;
878     size_t        nbytes;
879     int           s;

881     switch (valtype) {
882     case MOD_PROP_PERM:
883         nbytes = snprintf(buf, *bufsize, "%d", MOD_PROP_PERM_RW);
884         break;
885     case MOD_PROP_DEFAULT:
886     case MOD_PROP_POSSIBLE:
887         return (i_ipadm_get_prop(iph, arg, pdp, buf, bufsize,
888             proto, valtype));
889     case MOD_PROP_ACTIVE:
890         bzero(&lifr, sizeof (lifr));
891         (void) strcpy(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) strcpy(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 *pdp, 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) strcpy(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) strcpy(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_addrobj_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) strcpy(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) strcpy(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) strcpy(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) strcpy(mip->mpr_ifname, ifname,
1350             sizeof (mip->mpr_ifname));
1351     }

1353     (void) strcpy(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) strcpy(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 };
1854
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;
1865
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);
1869
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;
1909
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 }
1922
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/developer-dtrace.mf

1

26689 Sat Aug 18 10:37:05 2012

new/usr/src/pkg/manifests/developer-dtrace.mf

dccp: basic dtrace

```
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 set name=pkg.fmri value=pkg:/developer/dtrace@$(PKGVERS)
27 set name=pkg.description value="Dynamic Tracing (DTrace) Clients"
28 set name=pkg.summary value="DTrace Clients"
29 set name=info.classification \
30     value=org.opensolaris.category.2008:Development/System
31 set name=variant.arch value=$(ARCH)
32 dir path=usr group=sys
33 dir path=usr/demo
34 dir path=usr/demo/dtrace
35 dir path=usr/include
36 dir path=usr/include/sys
37 dir path=usr/lib
38 dir path=usr/lib/$(ARCH64)
39 dir path=usr/lib/devfsadm group=sys
40 dir path=usr/lib/devfsadm/linkmod group=sys
41 dir path=usr/lib/dtrace
42 dir path=usr/lib/dtrace/64
43 dir path=usr/lib/mdb group=sys
44 dir path=usr/lib/mdb/kvm group=sys
45 dir path=usr/lib/mdb/kvm/$(ARCH64) group=sys
46 dir path=usr/lib/mdb/raw group=sys
47 dir path=usr/lib/mdb/raw/$(ARCH64) group=sys
48 dir path=usr/sbin
49 dir path=usr/sbin/$(ARCH32)
50 dir path=usr/sbin/$(ARCH64)
51 dir path=usr/share
52 dir path=usr/share/lib
53 dir path=usr/share/lib/java group=sys
54 dir path=usr/share/lib/java/javadoc group=other
55 dir path=usr/share/lib/java/javadoc/dtrace group=other
56 dir path=usr/share/lib/java/javadoc/dtrace/api group=other
57 dir path=usr/share/lib/java/javadoc/dtrace/api/org group=other
58 dir path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris group=other
59 dir path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os group=other
60 dir path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace \
61     group=other
```

new/usr/src/pkg/manifests/developer-dtrace.mf

2

```
62 dir \
63     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
64     group=other
65 dir path=usr/share/lib/java/javadoc/dtrace/api/resources group=other
66 dir path=usr/share/lib/java/javadoc/dtrace/examples group=other
67 dir path=usr/share/lib/java/javadoc/dtrace/html group=other
68 dir path=usr/share/lib/java/javadoc/dtrace/images group=other
69 dir path=usr/share/man/man1m
70 dir path=usr/share/man/man3lib
71 file path=usr/demo/dtrace/applicat.d
72 file path=usr/demo/dtrace/badopen.d
73 file path=usr/demo/dtrace/begin.d
74 file path=usr/demo/dtrace/callout.d
75 file path=usr/demo/dtrace/clause.d
76 file path=usr/demo/dtrace/clear.d
77 file path=usr/demo/dtrace/countdown.d
78 file path=usr/demo/dtrace/counter.d
79 file path=usr/demo/dtrace/dateprof.d
80 file path=usr/demo/dtrace/delay.d
81 file path=usr/demo/dtrace/denorm.d
82 file path=usr/demo/dtrace/end.d
83 file path=usr/demo/dtrace/error.d
84 file path=usr/demo/dtrace/errorpath.d
85 file path=usr/demo/dtrace/find.d
86 file path=usr/demo/dtrace/firebird.d
87 file path=usr/demo/dtrace/hello.d
88 file path=usr/demo/dtrace/howlong.d
89 file path=usr/demo/dtrace/index.html
90 file path=usr/demo/dtrace/interp.d
91 file path=usr/demo/dtrace/interval.d
92 file path=usr/demo/dtrace/intr.d
93 file path=usr/demo/dtrace/iocpu.d
94 file path=usr/demo/dtrace/iosnoop.d
95 file path=usr/demo/dtrace/iotrough.d
96 file path=usr/demo/dtrace/iotime.d
97 file path=usr/demo/dtrace/ipio.d
98 file path=usr/demo/dtrace/ipproto.d
99 $(i386_ONLY)file path=usr/demo/dtrace/iprb.d
100 file path=usr/demo/dtrace/kstat.d
101 file path=usr/demo/dtrace/ksyms.d
102 file path=usr/demo/dtrace/libc.d
103 file path=usr/demo/dtrace/lquantize.d
104 file path=usr/demo/dtrace/lwptime.d
105 file path=usr/demo/dtrace/normalize.d
106 file path=usr/demo/dtrace/nscd.d
107 file path=usr/demo/dtrace/pri.d
108 file path=usr/demo/dtrace/printa.d
109 file path=usr/demo/dtrace/pritime.d
110 file path=usr/demo/dtrace/prof.d
111 file path=usr/demo/dtrace/profpri.d
112 file path=usr/demo/dtrace/proftime.d
113 file path=usr/demo/dtrace/putnext.d
114 file path=usr/demo/dtrace/qlen.d
115 file path=usr/demo/dtrace/qtime.d
116 file path=usr/demo/dtrace/renormalize.d
117 file path=usr/demo/dtrace/restest.d
118 file path=usr/demo/dtrace/ring.d
119 file path=usr/demo/dtrace/rtime.d
120 file path=usr/demo/dtrace/rwinfo.d
121 file path=usr/demo/dtrace/rwtime.d
122 file path=usr/demo/dtrace/sig.d
123 file path=usr/demo/dtrace/soffice.d
124 file path=usr/demo/dtrace/spec.d
125 file path=usr/demo/dtrace/specopen.d
126 file path=usr/demo/dtrace/ssd.d
127 file path=usr/demo/dtrace/syscall.d
```

```

128 file path=usr/demo/dtrace/tcplstbyte.d
129 file path=usr/demo/dtrace/tcpbytes.d
130 file path=usr/demo/dtrace/tcpbytesstat.d
131 file path=usr/demo/dtrace/tcpconnlst.d
132 file path=usr/demo/dtrace/tcpio.d
133 file path=usr/demo/dtrace/tcpioflags.d
134 file path=usr/demo/dtrace/tcprst.d
135 file path=usr/demo/dtrace/tcpsnoop.d
136 file path=usr/demo/dtrace/tcpstate.d
137 file path=usr/demo/dtrace/tcptop.d
138 file path=usr/demo/dtrace/tick.d
139 file path=usr/demo/dtrace/ticktime.d
140 file path=usr/demo/dtrace/time.d
141 file path=usr/demo/dtrace/tracewrite.d
142 file path=usr/demo/dtrace/trunc.d
143 file path=usr/demo/dtrace/trussrw.d
144 file path=usr/demo/dtrace/udpbytes.d
145 file path=usr/demo/dtrace/udpbytesstat.d
146 file path=usr/demo/dtrace/udpio.d
147 file path=usr/demo/dtrace/udpsnoop.d
148 file path=usr/demo/dtrace/udptop.d
149 file path=usr/demo/dtrace/userfunc.d
150 file path=usr/demo/dtrace/whatfor.d
151 file path=usr/demo/dtrace/whatlock.d
152 file path=usr/demo/dtrace/where.d
153 file path=usr/demo/dtrace/whererun.d
154 file path=usr/demo/dtrace/whoexec.d
155 file path=usr/demo/dtrace/whofer.d
156 file path=usr/demo/dtrace/whoio.d
157 file path=usr/demo/dtrace/whoempt.d
158 file path=usr/demo/dtrace/whoqueue.d
159 file path=usr/demo/dtrace/who steal.d
160 file path=usr/demo/dtrace/whowrite.d
161 file path=usr/demo/dtrace/writes.d
162 file path=usr/demo/dtrace/writesbycmd.d
163 file path=usr/demo/dtrace/writesbycmdfd.d
164 file path=usr/demo/dtrace/writetime.d
165 file path=usr/demo/dtrace/writetimeq.d
166 file path=usr/demo/dtrace/xioctl.d
167 file path=usr/demo/dtrace/xterm.d
168 file path=usr/demo/dtrace/xwork.d
169 file path=usr/include/dtrace.h
170 file path=usr/include/sys/dtrace.h
171 file path=usr/include/sys/dtrace_impl.h
172 file path=usr/include/sys/fasttrap.h
173 file path=usr/include/sys/fasttrap_impl.h
174 file path=usr/include/sys/fasttrap_isa.h
175 file path=usr/include/sys/lockstat.h
176 file path=usr/include/sys/sdt.h
177 file path=usr/lib/$(ARCH64)/libdtrace.so.1
178 file path=usr/lib/$(ARCH64)/libdtrace_jni.so.1
179 file path=usr/lib/$(ARCH64)/llib-ldtrace.ln
180 file path=usr/lib/devfsadm/linkmod/SUNW_dtrace_link.so group=sys
181 file path=usr/lib/dtrace/64/drti.o
182 file path=usr/lib/dtrace/dccp.d
183 #endif /* ! codereview */
184 file path=usr/lib/dtrace/drti.o
185 file path=usr/lib/dtrace/errno.d
186 file path=usr/lib/dtrace/fc.d
187 file path=usr/lib/dtrace/io.d
188 file path=usr/lib/dtrace/ip.d
189 file path=usr/lib/dtrace/iscsit.d
190 file path=usr/lib/dtrace/net.d
191 file path=usr/lib/dtrace/nfs.d
192 file path=usr/lib/dtrace/nfssrv.d
193 file path=usr/lib/dtrace/procfs.d

```

```

194 file path=usr/lib/dtrace/regs.d
195 file path=usr/lib/dtrace/sched.d
196 file path=usr/lib/dtrace/scsi.d
197 file path=usr/lib/dtrace/signal.d
198 file path=usr/lib/dtrace/srp.d
199 file path=usr/lib/dtrace/sysevent.d
200 file path=usr/lib/dtrace/tcp.d
201 file path=usr/lib/dtrace/udp.d
202 file path=usr/lib/dtrace/unistd.d
203 file path=usr/lib/libdtrace.so.1
204 file path=usr/lib/libdtrace_jni.so.1
205 file path=usr/lib/llib-ldtrace
206 file path=usr/lib/llib-ldtrace.ln
207 file path=usr/lib/mdb/kvm/$(ARCH64)/dtrace.so group=sys mode=0555
208 $(i386_ONLY)file path=usr/lib/mdb/kvm/dtrace.so group=sys mode=0555
209 file path=usr/lib/mdb/raw/$(ARCH64)/dof.so group=sys mode=0555
210 file path=usr/lib/mdb/raw/dof.so group=sys mode=0555
211 file path=usr/sbin/$(ARCH32)/dtrace mode=0555
212 $(i386_ONLY)file path=usr/sbin/$(ARCH32)/intrstat mode=0555
213 $(i386_ONLY)file path=usr/sbin/$(ARCH32)/lockstat mode=0555
214 $(i386_ONLY)file path=usr/sbin/$(ARCH32)/plockstat mode=0555
215 file path=usr/sbin/$(ARCH64)/dtrace mode=0555
216 file path=usr/sbin/$(ARCH64)/intrstat mode=0555
217 file path=usr/sbin/$(ARCH64)/lockstat mode=0555
218 file path=usr/sbin/$(ARCH64)/plockstat mode=0555
219 file path=usr/share/lib/java/javadoc/dtrace/api/allclasses-frame.html \
220 group=other
221 file path=usr/share/lib/java/javadoc/dtrace/api/allclasses-noframe.html \
222 group=other
223 file path=usr/share/lib/java/javadoc/dtrace/api/constant-values.html \
224 group=other
225 file path=usr/share/lib/java/javadoc/dtrace/api/deprecated-list.html \
226 group=other
227 file path=usr/share/lib/java/javadoc/dtrace/api/help-doc.html group=other
228 file path=usr/share/lib/java/javadoc/dtrace/api/index-all.html group=other
229 file path=usr/share/lib/java/javadoc/dtrace/api/index.html group=other
230 file \
231 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Aggrega
232 group=other
233 file \
234 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Aggrega
235 group=other
236 file \
237 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Aggrega
238 group=other
239 file \
240 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Aggrega
241 group=other
242 file \
243 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/AvgValu
244 group=other
245 file \
246 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
247 group=other
248 file \
249 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
250 group=other
251 file \
252 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
253 group=other
254 file \
255 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
256 group=other
257 file \
258 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
259

```

```
260     group=other
261 file \
262     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Consume
263     group=other
264 file \
265     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/CountVa
266     group=other
267 file \
268     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/DTraceE
269     group=other
270 file \
271     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/DataEve
272     group=other
273 file \
274     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Distrib
275     group=other
276 file \
277     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Distrib
278     group=other
279 file \
280     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Drop.Ki
281     group=other
282 file \
283     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Drop.ht
284     group=other
285 file \
286     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/DropEve
287     group=other
288 file \
289     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Error.h
290     group=other
291 file \
292     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ErrorEv
293     group=other
294 file \
295     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Excepti
296     group=other
297 file \
298     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ExitRec
299     group=other
300 file \
301     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Flow.Ki
302     group=other
303 file \
304     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Flow.ht
305     group=other
306 file \
307     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Interfa
308     group=other
309 file \
310     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Interfa
311     group=other
312 file \
313     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Interfa
314     group=other
315 file \
316     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Kernels
317     group=other
318 file \
319     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Kernels
320     group=other
321 file \
322     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/LinearD
323     group=other
324 file \
325     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/LocalCo
```

```
326     group=other
327 file \
328     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/LogDist
329     group=other
330 file \
331     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/LogLine
332     group=other
333 file \
334     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/MaxValu
335     group=other
336 file \
337     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/MinValu
338     group=other
339 file \
340     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Option.
341     group=other
342 file \
343     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/PrintaR
344     group=other
345 file \
346     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/PrintfR
347     group=other
348 file \
349     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Probe.h
350     group=other
351 file \
352     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ProbeDa
353     group=other
354 file \
355     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ProbeDa
356     group=other
357 file \
358     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ProbeDe
359     group=other
360 file \
361     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ProbeDe
362     group=other
363 file \
364     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ProbeIn
365     group=other
366 file \
367     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Process
368     group=other
369 file \
370     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Process
371     group=other
372 file \
373     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Process
374     group=other
375 file \
376     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Program
377     group=other
378 file \
379     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Program
380     group=other
381 file \
382     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Program
383     group=other
384 file \
385     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/Record.
386     group=other
387 file \
388     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/ScalarR
389     group=other
390 file \
391     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/StackFr
```



```
new/usr/src/pkg/manifests/developer-dtrace.mf
```

```

458 group=other
459 file \
460 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
461 group=other
462 file \
463 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
464 group=other
465 file \
466 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
467 group=other
468 file \
469 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
470 group=other
471 file \
472 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
473 group=other
474 file \
475 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
476 group=other
477 file \
478 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
479 group=other
480 file \
481 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
482 group=other
483 file \
484 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
485 group=other
486 file \
487 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
488 group=other
489 file \
490 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
491 group=other
492 file \
493 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
494 group=other
495 file \
496 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
497 group=other
498 file \
499 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
500 group=other
501 file \
502 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
503 group=other
504 file \
505 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
506 group=other
507 file \
508 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
509 group=other
510 file \
511 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
512 group=other
513 file \
514 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
515 group=other
516 file \
517 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
518 group=other
519 file \
520 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
521 group=other
522 file \
523 path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u

```

```

524     group=other
525 file \
526     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
527     group=other
528 file \
529     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
530     group=other
531 file \
532     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
533     group=other
534 file \
535     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
536     group=other
537 file \
538     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
539     group=other
540 file \
541     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
542     group=other
543 file \
544     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
545     group=other
546 file \
547     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
548     group=other
549 file \
550     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
551     group=other
552 file \
553     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
554     group=other
555 file \
556     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
557     group=other
558 file \
559     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
560     group=other
561 file \
562     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
563     group=other
564 file \
565     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
566     group=other
567 file \
568     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
569     group=other
570 file \
571     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
572     group=other
573 file \
574     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
575     group=other
576 file \
577     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
578     group=other
579 file \
580     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
581     group=other
582 file \
583     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
584     group=other
585 file \
586     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
587     group=other
588 file \
589     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u

```

```

590     group=other
591 file \
592     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
593     group=other
594 file \
595     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
596     group=other
597 file \
598     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
599     group=other
600 file \
601     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
602     group=other
603 file \
604     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
605     group=other
606 file \
607     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/class-u
608     group=other
609 file \
610     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/package
611     group=other
612 file \
613     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/package
614     group=other
615 file \
616     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/package
617     group=other
618 file \
619     path=usr/share/lib/java/javadoc/dtrace/api/org/opensolaris/os/dtrace/package
620     group=other
621 file path=usr/share/lib/java/javadoc/dtrace/api/overview-tree.html group=other
622 file path=usr/share/lib/java/javadoc/dtrace/api/package-list group=other
623 file path=usr/share/lib/java/javadoc/dtrace/api/resources/inherit.gif \
624     group=other
625 file path=usr/share/lib/java/javadoc/dtrace/api/serialized-form.html \
626     group=other
627 file path=usr/share/lib/java/javadoc/dtrace/api/stylesheets.css group=other
628 file path=usr/share/lib/java/javadoc/dtrace/examples/TestAPI.java group=other
629 file path=usr/share/lib/java/javadoc/dtrace/examples/TestAPI2.java group=other
630 file path=usr/share/lib/java/javadoc/dtrace/examples/TestTarget.java \
631     group=other
632 file path=usr/share/lib/java/javadoc/dtrace/examples/hello.d group=other
633 file path=usr/share/lib/java/javadoc/dtrace/examples/intrstat.d group=other
634 file path=usr/share/lib/java/javadoc/dtrace/examples/syscall.d group=other
635 file path=usr/share/lib/java/javadoc/dtrace/examples/target.d group=other
636 file path=usr/share/lib/java/javadoc/dtrace/html/JavaDTraceAPI.html \
637     group=other
638 file path=usr/share/lib/java/javadoc/dtrace/html/fast.html group=other
639 file path=usr/share/lib/java/javadoc/dtrace/images/JavaDTraceAPI.gif \
640     group=other
641 file path=usr/share/man/man1m/dtrace.1m
642 file path=usr/share/man/man1m/intrstat.1m
643 file path=usr/share/man/man1m/lockstat.1m
644 file path=usr/share/man/man1m/plockstat.1m
645 file path=usr/share/man/man3lib/libdtrace.3lib
646 hardlink path=usr/sbin/dtrace target=../usr/lib/isaexec
647 hardlink path=usr/sbin/intrstat target=../usr/lib/isaexec
648 hardlink path=usr/sbin/lockstat target=../usr/lib/isaexec
649 hardlink path=usr/sbin/plockstat target=../usr/lib/isaexec
650 legacy pkg=SUNWdtrc desc="Dynamic Tracing (DTrace) Clients" \
651     name="DTrace Clients"
652 license cr_Sun license=cr_Sun
653 license lic CDDL license=lic CDDL
654 link path=usr/lib/$(ARCH64)/libdtrace.so target=libdtrace.so.1
655 link path=usr/lib/$(ARCH64)/libdtrace_jni.so target=libdtrace_jni.so.1

```

```
new/usr/src/pkg/manifests/developer-dtrace.mf
```

11

```
656 link path=usr/lib/libdtrace.so target=libdtrace.so.1
```

```
657 link path=usr/lib/libdtrace_jni.so target=libdtrace_jni.so.1
```

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

1

```
*****
88501 Sat Aug 18 10:37:05 2012
new/usr/src/pkg/manifests/system-header.mf
dccc: small build fix
*****

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 2011 Nexenta Systems, Inc. All rights reserved.
24 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
25 #

27 set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
28 set name=pkg.description \
29     value="SunOS C/C++ header files for general development of software"
30 set name=pkg.summary value="SunOS Header Files"
31 set name=info.classification value=org.opensolaris.category.2008:System/Core
32 set name=variant.arch value=$(ARCH)
33 dir path=usr group=sys
34 dir path=usr/include
35 $(i386_ONLY)dir path=usr/include/$(ARCH64)
36 $(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
37 dir path=usr/include/arpa
38 dir path=usr/include/asm
39 dir path=usr/include/ast
40 dir path=usr/include/bsm
41 dir path=usr/include/dat
42 dir path=usr/include/des
43 dir path=usr/include/gssapi
44 dir path=usr/include/hal
45 $(i386_ONLY)dir path=usr/include/ia32
46 $(i386_ONLY)dir path=usr/include/ia32/sys
47 dir path=usr/include/inet
48 dir path=usr/include/inet/kssl
49 dir path=usr/include/ipp
50 dir path=usr/include/ipp/ipgpc
51 dir path=usr/include/iso
52 dir path=usr/include/kerberosv5
53 dir path=usr/include/libpolkit
54 dir path=usr/include/net
55 dir path=usr/include/netinet
56 dir path=usr/include/nfs
57 dir path=usr/include/protocols
58 dir path=usr/include/rpc
59 dir path=usr/include/rpcsvc
60 dir path=usr/include/sasl
61 dir path=usr/include/scsi
```

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

2

```
62 dir path=usr/include/scsi/plugins
63 dir path=usr/include/scsi/plugins/ses
64 dir path=usr/include/scsi/plugins/ses/framework
65 dir path=usr/include/scsi/plugins/ses/vendor
66 dir path=usr/include/scsi/plugins/smp
67 dir path=usr/include/scsi/plugins/smp/engine
68 dir path=usr/include/scsi/plugins/smp/framework
69 dir path=usr/include/security
70 dir path=usr/include/sharefs
71 dir path=usr/include/sys
72 dir path=usr/include/sys/av
73 dir path=usr/include/sys/contract
74 dir path=usr/include/sys/crypto
75 dir path=usr/include/sys/dktp
76 dir path=usr/include/sys/fc4
77 dir path=usr/include/sys/fm
78 dir path=usr/include/sys/fm/cpu
79 dir path=usr/include/sys/fm/fs
80 dir path=usr/include/sys/fm/io
81 $(sparc_ONLY)dir path=usr/include/sys/fpu
82 dir path=usr/include/sys/fs
83 dir path=usr/include/sys/hotplug
84 dir path=usr/include/sys/hotplug/pci
85 dir path=usr/include/sys/ib
86 dir path=usr/include/sys/ib/adapters
87 dir path=usr/include/sys/ib/adapters/hermon
88 dir path=usr/include/sys/ib/adapters/tavor
89 dir path=usr/include/sys/ib/clients
90 dir path=usr/include/sys/ib/clients/ibd
91 dir path=usr/include/sys/ib/clients/of
92 dir path=usr/include/sys/ib/clients/of/rdma
93 dir path=usr/include/sys/ib/clients/of/sol_ofs
94 dir path=usr/include/sys/ib/clients/of/sol_ucma
95 dir path=usr/include/sys/ib/clients/of/sol_umad
96 dir path=usr/include/sys/ib/clients/of/sol_uverbs
97 dir path=usr/include/sys/ib/ibnmx
98 dir path=usr/include/sys/ib/ibt1
99 dir path=usr/include/sys/ib/ibt1/impl
100 dir path=usr/include/sys/ib/mgt
101 dir path=usr/include/sys/ib/mgt/ibmf
102 dir path=usr/include/sys/iso
103 dir path=usr/include/sys/lvm
104 dir path=usr/include/sys/pcmcia
105 dir path=usr/include/sys/proc
106 dir path=usr/include/sys/rsm
107 $(i386_ONLY)dir path=usr/include/sys/sata group=sys
108 dir path=usr/include/sys/scsi
109 dir path=usr/include/sys/scsi/adapters
110 dir path=usr/include/sys/scsi/conf
111 dir path=usr/include/sys/scsi/generic
112 dir path=usr/include/sys/scsi/impl
113 dir path=usr/include/sys/scsi/targets
114 dir path=usr/include/sys/sysevent
115 dir path=usr/include/sys/tsol
116 dir path=usr/include/tsol
117 dir path=usr/include/uuid
118 $(sparc_ONLY)dir path=usr/include/v7
119 $(sparc_ONLY)dir path=usr/include/v7/sys
120 $(sparc_ONLY)dir path=usr/include/v9
121 $(sparc_ONLY)dir path=usr/include/v9/sys
122 dir path=usr/include/vm
123 dir path=usr/platform group=sys
124 $(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
125 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
126 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300/include
127 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010 group=sys
```

```
128 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010/include
129 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T12 group=sys
130 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T4 group=sys
131 $(sparc_ONLY)dir path=usr/platform/SUNW,SPARC-Enterprise group=sys
132 $(sparc_ONLY)dir path=usr/platform/SUNW,Serverblade1 group=sys
133 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-100 group=sys
134 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1000 group=sys
135 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1500 group=sys
136 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-2500 group=sys
137 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire group=sys
138 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-15000 group=sys
139 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-280R group=sys
140 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-480R group=sys
141 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-880 group=sys
142 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V215 group=sys
143 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V240 group=sys
144 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V250 group=sys
145 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V440 group=sys
146 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V445 group=sys
147 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V490 group=sys
148 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V890 group=sys
149 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-2 group=sys
150 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-250 group=sys
151 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-4 group=sys
152 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise group=sys
153 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise-10000 group=sys
154 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40 group=sys
155 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60 group=sys
156 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIi-Netract group=sys
157 $(i386_ONLY)dir path=usr/platform/i86pc group=sys
158 $(i386_ONLY)dir path=usr/platform/i86pc/include
159 $(i386_ONLY)dir path=usr/platform/i86pc/include/sys
160 $(i386_ONLY)dir path=usr/platform/i86pc/include/vm
161 $(i386_ONLY)dir path=usr/platform/i86xpv group=sys
162 $(i386_ONLY)dir path=usr/platform/i86xpv/include
163 $(i386_ONLY)dir path=usr/platform/i86xpv/include/sys
164 $(i386_ONLY)dir path=usr/platform/i86xpv/include/vm
165 $(sparc_ONLY)dir path=usr/platform/sun4u group=sys
166 $(sparc_ONLY)dir path=usr/platform/sun4u/include
167 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys
168 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c
169 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/clients
170 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/misc
171 $(sparc_ONLY)dir path=usr/platform/sun4u/include/vm
172 $(sparc_ONLY)dir path=usr/platform/sun4v group=sys
173 $(sparc_ONLY)dir path=usr/platform/sun4v/include
174 $(sparc_ONLY)dir path=usr/platform/sun4v/include/sys
175 $(sparc_ONLY)dir path=usr/platform/sun4v/include/vm
176 dir path=usr/share
177 dir path=usr/share/man
178 dir path=usr/share/man/man3head
179 dir path=usr/share/man/man4
180 dir path=usr/share/man/man5
181 dir path=usr/share/man/man7i
182 dir path=usr/share/src group=sys
183 dir path=usr/share/src/uts
184 $(i386_ONLY)dir path=usr/share/src/uts/i86pc
185 $(i386_ONLY)dir path=usr/share/src/uts/i86xpv
186 $(sparc_ONLY)dir path=usr/share/src/uts/sun4u
187 $(sparc_ONLY)dir path=usr/share/src/uts/sun4v
188 dir path=usr/xpg4
189 dir path=usr/xpg4/include
190 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/kdi_regs.h
191 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privmregs.h
192 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privregs.h
193 file path=usr/include/aio.h
```

```
194 file path=usr/include/alloca.h
195 file path=usr/include/apprace.h
196 file path=usr/include/apprace_impl.h
197 file path=usr/include/ar.h
198 file path=usr/include/archives.h
199 file path=usr/include/arpa/ftp.h
200 file path=usr/include/arpa/inet.h
201 file path=usr/include/arpa/nameser.h
202 file path=usr/include/arpa/nameser_compat.h
203 file path=usr/include/arpa/telnet.h
204 file path=usr/include/arpa/tftp.h
205 $(i386_ONLY)file path=usr/include/asm/atomic.h
206 $(i386_ONLY)file path=usr/include/asm/bitmap.h
207 $(i386_ONLY)file path=usr/include/asm/byteorder.h
208 $(i386_ONLY)file path=usr/include/asm/clock.h
209 $(i386_ONLY)file path=usr/include/asm/cpu.h
210 $(i386_ONLY)file path=usr/include/asm/cpuvar.h
211 $(sparc_ONLY)file path=usr/include/asm/flush.h
212 $(i386_ONLY)file path=usr/include/asm/htable.h
213 $(i386_ONLY)file path=usr/include/asm/mmu.h
214 file path=usr/include/asm/sunddi.h
215 file path=usr/include/asm/thread.h
216 file path=usr/include/assert.h
217 file path=usr/include/ast/align.h
218 file path=usr/include/ast/ast.h
219 file path=usr/include/ast/ast_botch.h
220 file path=usr/include/ast/ast_ccode.h
221 file path=usr/include/ast/ast_common.h
222 file path=usr/include/ast/ast_dir.h
223 file path=usr/include/ast/ast_dirent.h
224 file path=usr/include/ast/ast_fcntl.h
225 file path=usr/include/ast/ast_float.h
226 file path=usr/include/ast/ast_fs.h
227 file path=usr/include/ast/ast_getopt.h
228 file path=usr/include/ast/ast_icnv.h
229 file path=usr/include/ast/ast_lib.h
230 file path=usr/include/ast/ast_limits.h
231 file path=usr/include/ast/ast_map.h
232 file path=usr/include/ast/ast_mmap.h
233 file path=usr/include/ast/ast_mode.h
234 file path=usr/include/ast/ast_namval.h
235 file path=usr/include/ast/ast_ndbm.h
236 file path=usr/include/ast/ast_nl_types.h
237 file path=usr/include/ast/ast_param.h
238 file path=usr/include/ast/ast_standards.h
239 file path=usr/include/ast/ast_std.h
240 file path=usr/include/ast/ast_stdio.h
241 file path=usr/include/ast/ast_sys.h
242 file path=usr/include/ast/ast_time.h
243 file path=usr/include/ast/ast_tty.h
244 file path=usr/include/ast/ast_version.h
245 file path=usr/include/ast/ast_vfork.h
246 file path=usr/include/ast/ast_wait.h
247 file path=usr/include/ast/ast_wchar.h
248 file path=usr/include/ast/ast_windows.h
249 file path=usr/include/ast/bytesex.h
250 file path=usr/include/ast/ccode.h
251 file path=usr/include/ast/cdt.h
252 file path=usr/include/ast/cmd.h
253 file path=usr/include/ast/cmdext.h
254 file path=usr/include/ast/debug.h
255 file path=usr/include/ast/dirent.h
256 file path=usr/include/ast/dlldefs.h
257 file path=usr/include/ast/dt.h
258 file path=usr/include/ast/endian.h
259 file path=usr/include/ast/error.h
```

```
260 file path=usr/include/ast/find.h
261 file path=usr/include/ast/fnmatch.h
262 file path=usr/include/ast/fnv.h
263 file path=usr/include/ast/fs3d.h
264 file path=usr/include/ast/fts.h
265 file path=usr/include/ast/ftw.h
266 file path=usr/include/ast/ftwalk.h
267 file path=usr/include/ast/getopt.h
268 file path=usr/include/ast/glob.h
269 file path=usr/include/ast/hash.h
270 file path=usr/include/ast/hashkey.h
271 file path=usr/include/ast/hashpart.h
272 file path=usr/include/ast/history.h
273 file path=usr/include/ast/iconv.h
274 file path=usr/include/ast/ip6.h
275 file path=usr/include/ast/lc.h
276 file path=usr/include/ast/lstat.h
277 file path=usr/include/ast/magic.h
278 file path=usr/include/ast/magicid.h
279 file path=usr/include/ast/mc.h
280 file path=usr/include/ast/mime.h
281 file path=usr/include/ast/mnt.h
282 file path=usr/include/ast/modecanon.h
283 file path=usr/include/ast/modex.h
284 file path=usr/include/ast/namval.h
285 file path=usr/include/ast/nl_types.h
286 file path=usr/include/ast/nval.h
287 file path=usr/include/ast/option.h
288 file path=usr/include/ast/preroot.h
289 file path=usr/include/ast/proc.h
290 file path=usr/include/ast/prototyped.h
291 file path=usr/include/ast/re_comp.h
292 file path=usr/include/ast/recfmt.h
293 file path=usr/include/ast/regex.h
294 file path=usr/include/ast/regexp.h
295 file path=usr/include/ast/sfdisc.h
296 file path=usr/include/ast/sfio.h
297 file path=usr/include/ast/sfio_s.h
298 file path=usr/include/ast/sfio_t.h
299 file path=usr/include/ast/shcmd.h
300 file path=usr/include/ast/shell.h
301 file path=usr/include/ast/sig.h
302 file path=usr/include/ast/stack.h
303 file path=usr/include/ast/stak.h
304 file path=usr/include/ast/stdio.h
305 file path=usr/include/ast/stk.h
306 file path=usr/include/ast/sum.h
307 file path=usr/include/ast/swap.h
308 file path=usr/include/ast/tar.h
309 file path=usr/include/ast/times.h
310 file path=usr/include/ast/tm.h
311 file path=usr/include/ast/tmx.h
312 file path=usr/include/ast/tok.h
313 file path=usr/include/ast/tv.h
314 file path=usr/include/ast/usage.h
315 file path=usr/include/ast/vdb.h
316 file path=usr/include/ast/vecargs.h
317 file path=usr/include/ast/vmalloc.h
318 file path=usr/include/ast/wait.h
319 file path=usr/include/ast/wchar.h
320 file path=usr/include/ast/wordexp.h
321 file path=usr/include/atomic.h
322 file path=usr/include/attr.h
323 file path=usr/include/auth_attr.h
324 file path=usr/include/bsm/adt.h
325 file path=usr/include/bsm/adt_event.h
```

```
326 file path=usr/include/bsm/audit.h
327 file path=usr/include/bsm/audit_kernel.h
328 file path=usr/include/bsm/audit_kevents.h
329 file path=usr/include/bsm/audit_record.h
330 file path=usr/include/bsm/audit_uevents.h
331 file path=usr/include/bsm/devices.h
332 file path=usr/include/bsm/libbsm.h
333 file path=usr/include/config_admin.h
334 file path=usr/include/cpio.h
335 file path=usr/include/crypt.h
336 file path=usr/include/cryptoutil.h
337 file path=usr/include/ctype.h
338 file path=usr/include/curses.h
339 file path=usr/include/dat/dat.h
340 file path=usr/include/dat/dat_error.h
341 file path=usr/include/dat/dat_platform_specific.h
342 file path=usr/include/dat/dat_redirection.h
343 file path=usr/include/dat/dat_registry.h
344 file path=usr/include/dat/dat_vendor_specific.h
345 file path=usr/include/dat/udat.h
346 file path=usr/include/dat/udat_config.h
347 file path=usr/include/dat/udat_redirection.h
348 file path=usr/include/dat/udat_vendor_specific.h
349 file path=usr/include/deflt.h
350 file path=usr/include/des/des.h
351 file path=usr/include/des/desdata.h
352 file path=usr/include/des/softdes.h
353 file path=usr/include/device_info.h
354 file path=usr/include/devid.h
355 file path=usr/include/devmgmt.h
356 file path=usr/include/devpoll.h
357 file path=usr/include/dial.h
358 file path=usr/include/dirent.h
359 file path=usr/include/dlfcn.h
360 file path=usr/include/door.h
361 file path=usr/include/elf.h
362 file path=usr/include/err.h
363 file path=usr/include/errno.h
364 file path=usr/include/eti.h
365 file path=usr/include/euc.h
366 file path=usr/include/exacct.h
367 file path=usr/include/exacct_impl.h
368 file path=usr/include/exec_attr.h
369 file path=usr/include/execinfo.h
370 file path=usr/include/fatal.h
371 file path=usr/include/fcntl.h
372 file path=usr/include/float.h
373 file path=usr/include/fmtmsg.h
374 file path=usr/include/fnmatch.h
375 file path=usr/include/form.h
376 file path=usr/include/ftw.h
377 file path=usr/include/gelf.h
378 file path=usr/include/getopt.h
379 file path=usr/include/getwidth.h
380 file path=usr/include/glob.h
381 file path=usr/include/grp.h
382 file path=usr/include/gssapi/gssapi.h
383 file path=usr/include/gssapi/gssapi_ext.h
384 file path=usr/include/hal/libhal-storage.h
385 file path=usr/include/hal/libhal.h
386 $(i386_ONLY)file path=usr/include/ia32/sys/asm_linkage.h
387 $(i386_ONLY)file path=usr/include/ia32/sys/kdi_regs.h
388 $(i386_ONLY)file path=usr/include/ia32/sys/machtypes.h
389 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
390 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
391 $(i386_ONLY)file path=usr/include/ia32/sys/psw.h
```

```

392 $(i386_ONLY)file path=usr/include/ia32/sys/pte.h
393 $(i386_ONLY)file path=usr/include/ia32/sys/reg.h
394 $(i386_ONLY)file path=usr/include/ia32/sys/stack.h
395 $(i386_ONLY)file path=usr/include/ia32/sys/trap.h
396 $(i386_ONLY)file path=usr/include/ia32/sys/traptrace.h
397 file path=usr/include/iconv.h
398 file path=usr/include/idmap.h
399 file path=usr/include/ieeeep.h
400 file path=usr/include/ifaddrs.h
401 file path=usr/include/inet/arp.h
402 file path=usr/include/inet/common.h
403 file path=usr/include/inet/dccp.h
404 file path=usr/include/inet/dccp_ip.h
405 file path=usr/include/inet/dccp_stack.h
406 file path=usr/include/inet/dccp_stats.h
407 #endif /* ! codereview */
408 file path=usr/include/inet/ip.h
409 file path=usr/include/inet/ip6.h
410 file path=usr/include/inet/ip6_asp.h
411 file path=usr/include/inet/ip_arp.h
412 file path=usr/include/inet/ip_ftable.h
413 file path=usr/include/inet/ip_if.h
414 file path=usr/include/inet/ip_ire.h
415 file path=usr/include/inet/ip_multi.h
416 file path=usr/include/inet/ip_netinfo.h
417 file path=usr/include/inet/ip_rts.h
418 file path=usr/include/inet/ip_stack.h
419 file path=usr/include/inet/ipclassifier.h
420 file path=usr/include/inet/ipdrop.h
421 file path=usr/include/inet/ipnet.h
422 file path=usr/include/inet/ipp_common.h
423 file path=usr/include/inet/kssl/ksslapl.h
424 file path=usr/include/inet/led.h
425 file path=usr/include/inet/mi.h
426 file path=usr/include/inet/mib2.h
427 file path=usr/include/inet/nd.h
428 file path=usr/include/inet/optcom.h
429 file path=usr/include/inet/sctp_itf.h
430 file path=usr/include/inet/snmpcom.h
431 file path=usr/include/inet/tcp.h
432 file path=usr/include/inet/tcp_sack.h
433 file path=usr/include/inet/tcp_stack.h
434 file path=usr/include/inet/tcp_stats.h
435 file path=usr/include/inet/tunables.h
436 file path=usr/include/inet/wifi_ioctl.h
437 file path=usr/include/inttypes.h
438 file path=usr/include/imp.h
439 file path=usr/include/imp_admin.h
440 file path=usr/include/imp_mpathd.h
441 file path=usr/include/imp_query.h
442 file path=usr/include/ipp/ipgpc/ipgpc.h
443 file path=usr/include/ipp/ipp.h
444 file path=usr/include/ipp/ipp_config.h
445 file path=usr/include/ipp/ipp_impl.h
446 file path=usr/include/ipp/ippctl.h
447 file path=usr/include/iso/ctype_c99.h
448 file path=usr/include/iso/ctype_iso.h
449 file path=usr/include/iso/limits_iso.h
450 file path=usr/include/iso/locale_iso.h
451 file path=usr/include/iso/setjmp_iso.h
452 file path=usr/include/iso/signal_iso.h
453 file path=usr/include/iso/stdarg_c99.h
454 file path=usr/include/iso/stdarg_iso.h
455 file path=usr/include/iso/stddef_iso.h
456 file path=usr/include/iso/stdio_c99.h
457 file path=usr/include/iso/stdio_iso.h

```

```

458 file path=usr/include/iso/stdlib_c99.h
459 file path=usr/include/iso/stdlib_iso.h
460 file path=usr/include/iso/string_iso.h
461 file path=usr/include/iso/time_iso.h
462 file path=usr/include/iso/wchar_c99.h
463 file path=usr/include/iso/wchar_iso.h
464 file path=usr/include/iso/wctype_c99.h
465 file path=usr/include/iso/wctype_iso.h
466 file path=usr/include/iso646.h
467 file path=usr/include/kerberosv5/com_err.h
468 file path=usr/include/kerberosv5/krb5.h
469 file path=usr/include/kerberosv5/mit-sipb-copyright.h
470 file path=usr/include/kerberosv5/mit_copyright.h
471 file path=usr/include/klpd.h
472 file path=usr/include/kmfapi.h
473 file path=usr/include/kmftypes.h
474 file path=usr/include/kstat.h
475 file path=usr/include/kvm.h
476 file path=usr/include/langinfo.h
477 file path=usr/include/lastlog.h
478 file path=usr/include/lber.h
479 file path=usr/include/ldap.h
480 file path=usr/include/libcontract.h
481 file path=usr/include/libctf.h
482 file path=usr/include/libdevice.h
483 file path=usr/include/libdevinfo.h
484 file path=usr/include/libdldm.h
485 file path=usr/include/libdlbridge.h
486 file path=usr/include/libdlib.h
487 file path=usr/include/libdlink.h
488 file path=usr/include/libdlpi.h
489 file path=usr/include/libdlvlan.h
490 file path=usr/include/libelf.h
491 $(i386_ONLY)file path=usr/include/libfdisk.h
492 file path=usr/include/libfstyp.h
493 file path=usr/include/libfstyp_module.h
494 file path=usr/include/libgen.h
495 file path=usr/include/libgrubmgmt.h
496 file path=usr/include/libintl.h
497 file path=usr/include/libipmi.h
498 file path=usr/include/libipp.h
499 file path=usr/include/libnvpair.h
500 file path=usr/include/libnwam.h
501 file path=usr/include/libpolkit/libpolkit.h
502 file path=usr/include/librcm.h
503 file path=usr/include/libscf.h
504 file path=usr/include/libscf_priv.h
505 file path=usr/include/libshare.h
506 file path=usr/include/libsvm.h
507 file path=usr/include/libsysevent.h
508 file path=usr/include/libsysevent_impl.h
509 file path=usr/include/libtsnet.h
510 $(sparc_ONLY)file path=usr/include/libv12n.h
511 file path=usr/include/libw.h
512 file path=usr/include/libzfs.h
513 file path=usr/include/libzoneinfo.h
514 file path=usr/include/limits.h
515 file path=usr/include/linenum.h
516 file path=usr/include/link.h
517 file path=usr/include/listen.h
518 file path=usr/include/locale.h
519 file path=usr/include/macros.h
520 file path=usr/include/maillock.h
521 file path=usr/include/malloc.h
522 file path=usr/include/md4.h
523 file path=usr/include/md5.h

```

```
524 file path=usr/include/mdio.h
525 file path=usr/include/mdmn_changelog.h
526 file path=usr/include/memory.h
527 file path=usr/include/menu.h
528 file path=usr/include/meta.h
529 file path=usr/include/meta_basic.h
530 file path=usr/include/meta_runtime.h
531 file path=usr/include/metacl.h
532 file path=usr/include/metad.h
533 file path=usr/include/metadyn.h
534 file path=usr/include/metamed.h
535 file path=usr/include/metamhd.h
536 file path=usr/include/mhdx.h
537 file path=usr/include/mon.h
538 file path=usr/include/monetary.h
539 file path=usr/include/mp.h
540 file path=usr/include/mqueue.h
541 file path=usr/include/mtmalloc.h
542 file path=usr/include/nan.h
543 file path=usr/include/ndbm.h
544 file path=usr/include/ndpd.h
545 file path=usr/include/net/af.h
546 file path=usr/include/net/bridge.h
547 file path=usr/include/net/if.h
548 file path=usr/include/net/if_arp.h
549 file path=usr/include/net/if_dl.h
550 file path=usr/include/net/if_types.h
551 file path=usr/include/net/pfkeyv2.h
552 file path=usr/include/net/pfpolicy.h
553 file path=usr/include/net/ppp-comp.h
554 file path=usr/include/net/ppp_defs.h
555 file path=usr/include/net/pppio.h
556 file path=usr/include/net/radix.h
557 file path=usr/include/net/route.h
558 file path=usr/include/net/trill.h
559 file path=usr/include/net/vjcompress.h
560 file path=usr/include/netconfig.h
561 file path=usr/include/netdb.h
562 file path=usr/include/netdir.h
563 file path=usr/include/netinet/arp.h
564 file path=usr/include/netinet/dccp.h
565 #endif /* ! codereview */
566 file path=usr/include/netinet/dhcp.h
567 file path=usr/include/netinet/dhcp6.h
568 file path=usr/include/netinet/icmp6.h
569 file path=usr/include/netinet/icmp_var.h
570 file path=usr/include/netinet/if_ether.h
571 file path=usr/include/netinet/igmp.h
572 file path=usr/include/netinet/igmp_var.h
573 file path=usr/include/netinet/in.h
574 file path=usr/include/netinet/in_pcb.h
575 file path=usr/include/netinet/in_sysm.h
576 file path=usr/include/netinet/in_var.h
577 file path=usr/include/netinet/ip.h
578 file path=usr/include/netinet/ip6.h
579 file path=usr/include/netinet/ip_icmp.h
580 file path=usr/include/netinet/ip_mroute.h
581 file path=usr/include/netinet/ip_var.h
582 file path=usr/include/netinet/pim.h
583 file path=usr/include/netinet/sctp.h
584 file path=usr/include/netinet/tcp.h
585 file path=usr/include/netinet/tcp_debug.h
586 file path=usr/include/netinet/tcp_fsm.h
587 file path=usr/include/netinet/tcp_seq.h
588 file path=usr/include/netinet/tcp_timer.h
589 file path=usr/include/netinet/tcp_var.h
```

```
590 file path=usr/include/netinet/tcpip.h
591 file path=usr/include/netinet/udp.h
592 file path=usr/include/netinet/udp_var.h
593 file path=usr/include/netinet/vrrp.h
594 file path=usr/include/nfs/auth.h
595 file path=usr/include/nfs/export.h
596 file path=usr/include/nfs/lm.h
597 file path=usr/include/nfs/mapid.h
598 file path=usr/include/nfs/mount.h
599 file path=usr/include/nfs/nfs.h
600 file path=usr/include/nfs/nfs4.h
601 file path=usr/include/nfs/nfs4_attr.h
602 file path=usr/include/nfs/nfs4_clnt.h
603 file path=usr/include/nfs/nfs4_db_impl.h
604 file path=usr/include/nfs/nfs4_idmap_impl.h
605 file path=usr/include/nfs/nfs4_kprot.h
606 file path=usr/include/nfs/nfs_acl.h
607 file path=usr/include/nfs/nfs_clnt.h
608 file path=usr/include/nfs/nfs_cmd.h
609 file path=usr/include/nfs/nfs_log.h
610 file path=usr/include/nfs/nfs_sec.h
611 file path=usr/include/nfs/nfsid_map.h
612 file path=usr/include/nfs/nfssys.h
613 file path=usr/include/nfs/rnode.h
614 file path=usr/include/nfs/rnode4.h
615 file path=usr/include/nl_types.h
616 file path=usr/include/nlist.h
617 file path=usr/include/note.h
618 file path=usr/include/nss_common.h
619 file path=usr/include/nss_dbdefs.h
620 file path=usr/include/nss_netdir.h
621 file path=usr/include/nsswitch.h
622 file path=usr/include/panel.h
623 file path=usr/include/paths.h
624 file path=usr/include/pcsample.h
625 file path=usr/include/pfmt.h
626 file path=usr/include/pkgdev.h
627 file path=usr/include/pkginfo.h
628 file path=usr/include/pkglocs.h
629 file path=usr/include/pkgstrct.h
630 file path=usr/include/pkgtrans.h
631 file path=usr/include/poll.h
632 file path=usr/include/port.h
633 file path=usr/include/priv.h
634 file path=usr/include/proc_service.h
635 file path=usr/include/procfs.h
636 file path=usr/include/prof.h
637 file path=usr/include/prof_attr.h
638 file path=usr/include/project.h
639 file path=usr/include/protocols/dumpprestore.h
640 file path=usr/include/protocols/routed.h
641 file path=usr/include/protocols/rwhod.h
642 file path=usr/include/protocols/timed.h
643 file path=usr/include/pthread.h
644 file path=usr/include/pw.h
645 file path=usr/include/pwd.h
646 file path=usr/include/rcm_module.h
647 file path=usr/include/rctl.h
648 file path=usr/include/re_comp.h
649 file path=usr/include/regex.h
650 file path=usr/include/regexp.h
651 file path=usr/include/regexpr.h
652 file path=usr/include/resolv.h
653 file path=usr/include/rje.h
654 file path=usr/include/rp_plugin.h
655 file path=usr/include/rpc/auth.h
```



```
656 file path=usr/include/rpc/auth_des.h
657 file path=usr/include/rpc/auth_sys.h
658 file path=usr/include/rpc/auth_unix.h
659 file path=usr/include/rpc/bootparam.h
660 file path=usr/include/rpc/clnt.h
661 file path=usr/include/rpc/clnt_soc.h
662 file path=usr/include/rpc/clnt_stat.h
663 file path=usr/include/rpc/des_crypt.h
664 $(sparc_ONLY)file path=usr/include/rpc/ib.h
665 file path=usr/include/rpc/key_prot.h
666 file path=usr/include/rpc/nettype.h
667 file path=usr/include/rpc/pmap_clnt.h
668 file path=usr/include/rpc/pmap_prot.h
669 file path=usr/include/rpc/pmap_prot.x
670 file path=usr/include/rpc/pmap_rmt.h
671 file path=usr/include/rpc/raw.h
672 file path=usr/include/rpc/rpc.h
673 file path=usr/include/rpc/rpc_com.h
674 file path=usr/include/rpc/rpc_msg.h
675 file path=usr/include/rpc/rpc_rdma.h
676 file path=usr/include/rpc/rpc_sztypes.h
677 file path=usr/include/rpc/rpcb_clnt.h
678 file path=usr/include/rpc/rpcb_prot.h
679 file path=usr/include/rpc/rpcb_prot.x
680 file path=usr/include/rpc/rpcent.h
681 file path=usr/include/rpc/rpcsec_gss.h
682 file path=usr/include/rpc/rpcsys.h
683 file path=usr/include/rpc/svc.h
684 file path=usr/include/rpc/svc_auth.h
685 file path=usr/include/rpc/svc_mt.h
686 file path=usr/include/rpc/svc_soc.h
687 file path=usr/include/rpc/types.h
688 file path=usr/include/rpc/xdr.h
689 file path=usr/include/rpcsvc/autofs_prot.h
690 file path=usr/include/rpcsvc/autofs_prot.x
691 file path=usr/include/rpcsvc/bootparam.h
692 file path=usr/include/rpcsvc/bootparam_prot.h
693 file path=usr/include/rpcsvc/bootparam_prot.x
694 file path=usr/include/rpcsvc/dbm.h
695 file path=usr/include/rpcsvc/key_prot.x
696 file path=usr/include/rpcsvc/mount.h
697 file path=usr/include/rpcsvc/mount.x
698 file path=usr/include/rpcsvc/nfs4_prot.h
699 file path=usr/include/rpcsvc/nfs4_prot.x
700 file path=usr/include/rpcsvc/nfs_acl.h
701 file path=usr/include/rpcsvc/nfs_acl.x
702 file path=usr/include/rpcsvc/nfs_prot.h
703 file path=usr/include/rpcsvc/nfs_prot.x
704 file path=usr/include/rpcsvc/nis.h
705 file path=usr/include/rpcsvc/nis.x
706 file path=usr/include/rpcsvc/nis_db.h
707 file path=usr/include/rpcsvc/nis_object.x
708 file path=usr/include/rpcsvc/nislib.h
709 file path=usr/include/rpcsvc/nlm_prot.h
710 file path=usr/include/rpcsvc/nlm_prot.x
711 file path=usr/include/rpcsvc/nsm_addr.h
712 file path=usr/include/rpcsvc/nsm_addr.x
713 file path=usr/include/rpcsvc/rex.h
714 file path=usr/include/rpcsvc/rex.x
715 file path=usr/include/rpcsvc/rpc_sztypes.h
716 file path=usr/include/rpcsvc/rpc_sztypes.x
717 file path=usr/include/rpcsvc/rquota.h
718 file path=usr/include/rpcsvc/rquota.x
719 file path=usr/include/rpcsvc/rstat.h
720 file path=usr/include/rpcsvc/rstat.x
721 file path=usr/include/rpcsvc/rusers.h
```

```
722 file path=usr/include/rpcsvc/rusers.x
723 file path=usr/include/rpcsvc/rwall.h
724 file path=usr/include/rpcsvc/rwall.x
725 file path=usr/include/rpcsvc/sm_inter.h
726 file path=usr/include/rpcsvc/sm_inter.x
727 file path=usr/include/rpcsvc/spray.h
728 file path=usr/include/rpcsvc/spray.x
729 file path=usr/include/rpcsvc/ufs_prot.h
730 file path=usr/include/rpcsvc/ufs_prot.x
731 file path=usr/include/rpcsvc/yp.x
732 file path=usr/include/rpcsvc/yp_prot.h
733 file path=usr/include/rpcsvc/ypclnt.h
734 file path=usr/include/rpcsvc/yppasswd.h
735 file path=usr/include/rpcsvc/ypupd.h
736 file path=usr/include/rsmapi.h
737 file path=usr/include/rtld_db.h
738 file path=usr/include/sac.h
739 file path=usr/include/sasl/prop.h
740 file path=usr/include/sasl/sasl.h
741 file path=usr/include/sasl/saslplug.h
742 file path=usr/include/sasl/saslutil.h
743 file path=usr/include/sched.h
744 file path=usr/include/schedctl.h
745 file path=usr/include/scsi/libscsi.h
746 file path=usr/include/scsi/libses.h
747 file path=usr/include/scsi/libses_plugin.h
748 file path=usr/include/scsi/libsmpl.h
749 file path=usr/include/scsi/libsmpl_plugin.h
750 file path=usr/include/scsi/plugins/ses/framework/libses.h
751 file path=usr/include/scsi/plugins/ses/framework/ses2.h
752 file path=usr/include/scsi/plugins/ses/framework/ses2_impl.h
753 file path=usr/include/scsi/plugins/ses/vendor/sun.h
754 file path=usr/include/sdp.h
755 file path=usr/include/search.h
756 file path=usr/include/secdb.h
757 file path=usr/include/security/auditd.h
758 file path=usr/include/security/cryptoki.h
759 file path=usr/include/security/pam_appl.h
760 file path=usr/include/security/pam_modules.h
761 file path=usr/include/security/pkcs11.h
762 file path=usr/include/security/pkcs11f.h
763 file path=usr/include/security/pkcs11t.h
764 file path=usr/include/semaphore.h
765 file path=usr/include/setjmp.h
766 file path=usr/include/sgtty.h
767 file path=usr/include/sha1.h
768 file path=usr/include/sha2.h
769 file path=usr/include/shadow.h
770 file path=usr/include/sharefs/share.h
771 file path=usr/include/sharefs/sharefs.h
772 file path=usr/include/sharefs/sharetabs.h
773 file path=usr/include/siginfo.h
774 file path=usr/include/signal.h
775 file path=usr/include/sip.h
776 file path=usr/include/smbios.h
777 file path=usr/include/spawn.h
778 $(i386_ONLY)file path=usr/include/stack_unwind.h
779 file path=usr/include/stdarg.h
780 file path=usr/include/stdbool.h
781 file path=usr/include/stddef.h
782 file path=usr/include/stdint.h
783 file path=usr/include/stdio.h
784 file path=usr/include/stdio_ext.h
785 file path=usr/include/stdio_impl.h
786 file path=usr/include/stdio_tag.h
787 file path=usr/include/stdlib.h
```

```
788 file path=usr/include/storclass.h
789 file path=usr/include/string.h
790 file path=usr/include/strings.h
791 file path=usr/include/stropts.h
792 file path=usr/include/syms.h
793 file path=usr/include/synch.h
794 file path=usr/include/sys/acct.h
795 file path=usr/include/sys/acctctl.h
796 file path=usr/include/sys/acl.h
797 file path=usr/include/sys/acl_impl.h
798 file path=usr/include/sys/acpi_drv.h
799 file path=usr/include/sys/aio.h
800 file path=usr/include/sys/aio_impl.h
801 file path=usr/include/sys/aio_req.h
802 file path=usr/include/sys/aioch.h
803 file path=usr/include/sys/archsystem.h
804 file path=usr/include/sys/ascii.h
805 file path=usr/include/sys/asm_linkage.h
806 file path=usr/include/sys/asynch.h
807 file path=usr/include/sys/atomic.h
808 file path=usr/include/sys/attr.h
809 file path=usr/include/sys/autoconf.h
810 file path=usr/include/sys/auxv.h
811 file path=usr/include/sys/auxv_386.h
812 file path=usr/include/sys/auxv_SPARC.h
813 file path=usr/include/sys/av/iec61883.h
814 file path=usr/include/sys/avintr.h
815 file path=usr/include/sys/avl.h
816 file path=usr/include/sys/avl_impl.h
817 file path=usr/include/sys/bitmap.h
818 file path=usr/include/sys/bitset.h
819 file path=usr/include/sys/bl.h
820 file path=usr/include/sys/blkdev.h
821 file path=usr/include/sys/bmc_intf.h
822 file path=usr/include/sys/bofi.h
823 file path=usr/include/sys/bofi_impl.h
824 file path=usr/include/sys/bootconf.h
825 $(i386_ONLY)file path=usr/include/sys/bootregs.h
826 file path=usr/include/sys/bootstat.h
827 $(i386_ONLY)file path=usr/include/sys/bootsvcs.h
828 file path=usr/include/sys/bpp_io.h
829 file path=usr/include/sys/brand.h
830 file path=usr/include/sys/buf.h
831 file path=usr/include/sys/bufmod.h
832 file path=usr/include/sys/bustypes.h
833 file path=usr/include/sys/byteorder.h
834 file path=usr/include/sys/callb.h
835 file path=usr/include/sys/callo.h
836 file path=usr/include/sys/cap_util.h
837 file path=usr/include/sys/ccompile.h
838 file path=usr/include/sys/cdio.h
839 file path=usr/include/sys/cis.h
840 file path=usr/include/sys/cis_handlers.h
841 file path=usr/include/sys/cis_protos.h
842 file path=usr/include/sys/cladm.h
843 file path=usr/include/sys/class.h
844 file path=usr/include/sys/clconf.h
845 file path=usr/include/sys/cmlb.h
846 file path=usr/include/sys/cmn_err.h
847 $(sparc_ONLY)file path=usr/include/sys/cmpregs.h
848 file path=usr/include/sys/compress.h
849 file path=usr/include/sys/conddvar.h
850 file path=usr/include/sys/conddvar_impl.h
851 file path=usr/include/sys/conf.h
852 file path=usr/include/sys/consdev.h
853 file path=usr/include/sys/console.h
```

```
854 file path=usr/include/sys/consplat.h
855 file path=usr/include/sys/contract.h
856 file path=usr/include/sys/contract/device.h
857 file path=usr/include/sys/contract/device_impl.h
858 file path=usr/include/sys/contract/process.h
859 file path=usr/include/sys/contract/process_impl.h
860 file path=usr/include/sys/contract_impl.h
861 $(i386_ONLY)file path=usr/include/sys/controlregs.h
862 file path=usr/include/sys/copyops.h
863 file path=usr/include/sys/core.h
864 file path=usr/include/sys/corectl.h
865 file path=usr/include/sys/cpc_impl.h
866 file path=usr/include/sys/cpc_pcbe.h
867 file path=usr/include/sys/cpr.h
868 file path=usr/include/sys/cpu.h
869 file path=usr/include/sys/cpucaps.h
870 file path=usr/include/sys/cpucaps_impl.h
871 file path=usr/include/sys/cpupart.h
872 file path=usr/include/sys/cpuvvar.h
873 file path=usr/include/sys/crc32.h
874 file path=usr/include/sys/cred.h
875 file path=usr/include/sys/cred_impl.h
876 file path=usr/include/sys/crtctl.h
877 file path=usr/include/sys/crypto/api.h
878 file path=usr/include/sys/crypto/common.h
879 file path=usr/include/sys/crypto/ioctl.h
880 file path=usr/include/sys/crypto/ioctladmin.h
881 file path=usr/include/sys/crypto/spi.h
882 file path=usr/include/sys/cs.h
883 file path=usr/include/sys/cs_priv.h
884 file path=usr/include/sys/cs_strings.h
885 file path=usr/include/sys/cs_stubs.h
886 file path=usr/include/sys/cs_types.h
887 file path=usr/include/sys/csiioctl.h
888 file path=usr/include/sys/ctf.h
889 file path=usr/include/sys/ctf_api.h
890 file path=usr/include/sys/ctfs.h
891 file path=usr/include/sys/ctfs_impl.h
892 file path=usr/include/sys/ctype.h
893 file path=usr/include/sys/cyclic.h
894 file path=usr/include/sys/cyclic_impl.h
895 file path=usr/include/sys/dacf.h
896 file path=usr/include/sys/dacf_impl.h
897 file path=usr/include/sys/damap.h
898 file path=usr/include/sys/damap_impl.h
899 file path=usr/include/sys/dc_ki.h
900 file path=usr/include/sys/ddi.h
901 file path=usr/include/sys/ddi_hp.h
902 file path=usr/include/sys/ddi_hp_impl.h
903 file path=usr/include/sys/ddi_impldefs.h
904 file path=usr/include/sys/ddi_implfuncs.h
905 file path=usr/include/sys/ddi_intr.h
906 file path=usr/include/sys/ddi_intr_impl.h
907 file path=usr/include/sys/ddi_isa.h
908 file path=usr/include/sys/ddi_obsolete.h
909 file path=usr/include/sys/ddi_timer.h
910 file path=usr/include/sys/ddidevmap.h
911 file path=usr/include/sys/ddidmareq.h
912 file path=usr/include/sys/ddifm.h
913 file path=usr/include/sys/ddifm_impl.h
914 file path=usr/include/sys/ddimapreq.h
915 file path=usr/include/sys/ddipropdefs.h
916 file path=usr/include/sys/dditypes.h
917 file path=usr/include/sys/debug.h
918 $(i386_ONLY)file path=usr/include/sys/debugreg.h
919 file path=usr/include/sys/des.h
```

```
920 file path=usr/include/sys/devcache.h
921 file path=usr/include/sys/devcache_impl.h
922 file path=usr/include/sys/devctl.h
923 file path=usr/include/sys/devfm.h
924 file path=usr/include/sys/devid_cache.h
925 file path=usr/include/sys/devinfo_impl.h
926 file path=usr/include/sys/devops.h
927 file path=usr/include/sys/devpolicy.h
928 file path=usr/include/sys/devpoll.h
929 file path=usr/include/sys/dirent.h
930 file path=usr/include/sys/disp.h
931 file path=usr/include/sys/dkbad.h
932 file path=usr/include/sys/dkio.h
933 file path=usr/include/sys/dklabel.h
934 $(sparc_ONLY)file path=usr/include/sys/dkmpio.h
935 $(i386_ONLY)file path=usr/include/sys/dktp/altctr.h
936 $(i386_ONLY)file path=usr/include/sys/dktp/cmpkt.h
937 file path=usr/include/sys/dktp/dadkio.h
938 file path=usr/include/sys/dktp/fdisk.h
939 file path=usr/include/sys/dl.h
940 file path=usr/include/sys/dld.h
941 file path=usr/include/sys/dlpi.h
942 file path=usr/include/sys/dls_mgmt.h
943 $(i386_ONLY)file path=usr/include/sys/dma_engine.h
944 file path=usr/include/sys/dma_i8237A.h
945 file path=usr/include/sys/dnlc.h
946 file path=usr/include/sys/door.h
947 file path=usr/include/sys/door_data.h
948 file path=usr/include/sys/door_impl.h
949 file path=usr/include/sys/dumphdr.h
950 file path=usr/include/sys/ecppio.h
951 file path=usr/include/sys/ecppreg.h
952 file path=usr/include/sys/ecppsys.h
953 file path=usr/include/sys/ecppvar.h
954 file path=usr/include/sys/efi_partition.h
955 file path=usr/include/sys/elf.h
956 file path=usr/include/sys/elf_386.h
957 file path=usr/include/sys/elf_SPARC.h
958 file path=usr/include/sys/elf_amd64.h
959 file path=usr/include/sys/elf_notes.h
960 file path=usr/include/sys/elftypes.h
961 file path=usr/include/sys/epm.h
962 file path=usr/include/sys/errno.h
963 file path=usr/include/sys/errorq.h
964 file path=usr/include/sys/errorq_impl.h
965 file path=usr/include/sys/esunddi.h
966 file path=usr/include/sys/ethernet.h
967 file path=usr/include/sys/euc.h
968 file path=usr/include/sys/eucioctl.h
969 file path=usr/include/sys/exacct.h
970 file path=usr/include/sys/exacct_catalog.h
971 file path=usr/include/sys/exacct_impl.h
972 file path=usr/include/sys/exec.h
973 file path=usr/include/sys/exechdr.h
974 file path=usr/include/sys/fault.h
975 file path=usr/include/sys/fbio.h
976 file path=usr/include/sys/fbuf.h
977 file path=usr/include/sys/fc4/fc.h
978 file path=usr/include/sys/fc4/fc_transport.h
979 file path=usr/include/sys/fc4/fcal.h
980 file path=usr/include/sys/fc4/fcal_linkapp.h
981 file path=usr/include/sys/fc4/fcal_transport.h
982 file path=usr/include/sys/fc4/fcio.h
983 file path=usr/include/sys/fc4/fcp.h
984 file path=usr/include/sys/fc4/linkapp.h
985 file path=usr/include/sys/fcntl.h
```

```
986 file path=usr/include/sys/fdbuffer.h
987 file path=usr/include/sys/fdio.h
988 $(sparc_ONLY)file path=usr/include/sys/fdreg.h
989 $(sparc_ONLY)file path=usr/include/sys/fdvar.h
990 file path=usr/include/sys/feature_tests.h
991 file path=usr/include/sys/fem.h
992 file path=usr/include/sys/file.h
993 file path=usr/include/sys/filio.h
994 file path=usr/include/sys/flock.h
995 file path=usr/include/sys/flock_impl.h
996 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/SPARC64-VI.h
997 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-II.h
998 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-III.h
999 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-T1.h
1000 file path=usr/include/sys/fm/fs/zfs.h
1001 file path=usr/include/sys/fm/io/ddi.h
1002 file path=usr/include/sys/fm/io/disk.h
1003 file path=usr/include/sys/fm/io/opl_mc_fm.h
1004 file path=usr/include/sys/fm/io/pci.h
1005 file path=usr/include/sys/fm/io/scsi.h
1006 file path=usr/include/sys/fm/io/sun4upci.h
1007 file path=usr/include/sys/fm/protocol.h
1008 file path=usr/include/sys/fm/util.h
1009 file path=usr/include/sys/fork.h
1010 $(i386_ONLY)file path=usr/include/sys/fp.h
1011 $(sparc_ONLY)file path=usr/include/sys/fpu/fpu_simulator.h
1012 $(sparc_ONLY)file path=usr/include/sys/fpu/fpusystm.h
1013 $(sparc_ONLY)file path=usr/include/sys/fpu/globals.h
1014 $(sparc_ONLY)file path=usr/include/sys/fpu/ieee.h
1015 file path=usr/include/sys/frame.h
1016 file path=usr/include/sys/fs/autofs.h
1017 file path=usr/include/sys/fs/cacheofs_dir.h
1018 file path=usr/include/sys/fs/cacheofs_dlog.h
1019 file path=usr/include/sys/fs/cacheofs_filegrp.h
1020 file path=usr/include/sys/fs/cacheofs_fs.h
1021 file path=usr/include/sys/fs/cacheofs_fscache.h
1022 file path=usr/include/sys/fs/cacheofs_ioctl.h
1023 file path=usr/include/sys/fs/cacheofs_log.h
1024 file path=usr/include/sys/fs/decomp.h
1025 file path=usr/include/sys/fs/dv_node.h
1026 file path=usr/include/sys/fs/fifonode.h
1027 file path=usr/include/sys/fs/hsfs_isospec.h
1028 file path=usr/include/sys/fs/hsfs_node.h
1029 file path=usr/include/sys/fs/hsfs_rrip.h
1030 file path=usr/include/sys/fs/hsfs_spec.h
1031 file path=usr/include/sys/fs/hsfs_susp.h
1032 file path=usr/include/sys/fs/lofs_info.h
1033 file path=usr/include/sys/fs/lofs_node.h
1034 file path=usr/include/sys/fs/mntdata.h
1035 file path=usr/include/sys/fs/namenode.h
1036 file path=usr/include/sys/fs/pc_dir.h
1037 file path=usr/include/sys/fs/pc_fs.h
1038 file path=usr/include/sys/fs/pc_label.h
1039 file path=usr/include/sys/fs/pc_node.h
1040 file path=usr/include/sys/fs/pxfs_ki.h
1041 file path=usr/include/sys/fs/sdev_impl.h
1042 file path=usr/include/sys/fs/snode.h
1043 file path=usr/include/sys/fs/swapnode.h
1044 file path=usr/include/sys/fs/tmp.h
1045 file path=usr/include/sys/fs/tmpnode.h
1046 file path=usr/include/sys/fs/udf_inode.h
1047 file path=usr/include/sys/fs/udf_volume.h
1048 file path=usr/include/sys/fs/ufs_acl.h
1049 file path=usr/include/sys/fs/ufs_bio.h
1050 file path=usr/include/sys/fs/ufs_filio.h
1051 file path=usr/include/sys/fs/ufs_fs.h
```

```
1052 file path=usr/include/sys/fs/ufs_fsdir.h
1053 file path=usr/include/sys/fs/ufs_inode.h
1054 file path=usr/include/sys/fs/ufs_lockfs.h
1055 file path=usr/include/sys/fs/ufs_log.h
1056 file path=usr/include/sys/fs/ufs_mount.h
1057 file path=usr/include/sys/fs/ufs_panic.h
1058 file path=usr/include/sys/fs/ufs_prot.h
1059 file path=usr/include/sys/fs/ufs_quota.h
1060 file path=usr/include/sys/fs/ufs_snap.h
1061 file path=usr/include/sys/fs/ufs_trans.h
1062 file path=usr/include/sys/fs/zfs.h
1063 file path=usr/include/sys/fs_reparse.h
1064 file path=usr/include/sys/fs_subr.h
1065 file path=usr/include/sys/fsid.h
1066 $(sparc_ONLY)file path=usr/include/sys/fsr.h
1067 file path=usr/include/sys/fss.h
1068 file path=usr/include/sys/fssnap.h
1069 file path=usr/include/sys/fssnap_if.h
1070 file path=usr/include/sys/fsspriocntl.h
1071 file path=usr/include/sys/fstyp.h
1072 file path=usr/include/sys/ftrace.h
1073 file path=usr/include/sys/fx.h
1074 file path=usr/include/sys/fxpriocntl.h
1075 file path=usr/include/sys/gfs.h
1076 file path=usr/include/sys/gld.h
1077 file path=usr/include/sys/gldpriv.h
1078 file path=usr/include/sys/group.h
1079 file path=usr/include/sys/hdio.h
1080 file path=usr/include/sys/hook.h
1081 file path=usr/include/sys/hook_event.h
1082 file path=usr/include/sys/hook_impl.h
1083 file path=usr/include/sys/hotplug/hpcsvc.h
1084 file path=usr/include/sys/hotplug/hpctrl.h
1085 file path=usr/include/sys/hotplug/pci/pcicfg.h
1086 file path=usr/include/sys/hotplug/pci/pcihp.h
1087 file path=usr/include/sys/hwconf.h
1088 $(i386_ONLY)file path=usr/include/sys/hypervisor.h
1089 $(i386_ONLY)file path=usr/include/sys/i8272A.h
1090 file path=usr/include/sys/ia.h
1091 file path=usr/include/sys/iapriocntl.h
1092 file path=usr/include/sys/ib/adapters/hermon/hermon_ioctl.h
1093 file path=usr/include/sys/ib/adapters/mlnx_umap.h
1094 file path=usr/include/sys/ib/adapters/tavor/tavor_ioctl.h
1095 file path=usr/include/sys/ib/clients/ibd/ibd.h
1096 file path=usr/include/sys/ib/clients/of/ofa_solaris.h
1097 file path=usr/include/sys/ib/clients/of/ofed_kernel.h
1098 file path=usr/include/sys/ib/clients/of/rdma/ib_addr.h
1099 file path=usr/include/sys/ib/clients/of/rdma/ib_user_mad.h
1100 file path=usr/include/sys/ib/clients/of/rdma/ib_user_sa.h
1101 file path=usr/include/sys/ib/clients/of/rdma/ib_user_verbs.h
1102 file path=usr/include/sys/ib/clients/of/rdma/ib_verbs.h
1103 file path=usr/include/sys/ib/clients/of/rdma/rdma_cm.h
1104 file path=usr/include/sys/ib/clients/of/rdma/rdma_user_cm.h
1105 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_cma.h
1106 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ib_cma.h
1107 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_kverb_impl.h
1108 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ofs_common.h
1109 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_rdma_user_cm.h
1110 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_ucma.h
1111 file path=usr/include/sys/ib/clients/of/sol_umad/sol_umad.h
1112 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs.h
1113 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs2ucma.h
1114 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_comp.h
1115 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_event.h
1116 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_hca.h
1117 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_qp.h
```

```
1118 file path=usr/include/sys/ib/ib_pkt_hdrs.h
1119 file path=usr/include/sys/ib/ib_types.h
1120 file path=usr/include/sys/ib/ibnexus/ibnexus_devctl.h
1121 file path=usr/include/sys/ib/ibtl/ibci.h
1122 file path=usr/include/sys/ib/ibtl/ibti.h
1123 file path=usr/include/sys/ib/ibtl/ibti_cm.h
1124 file path=usr/include/sys/ib/ibtl/ibti_common.h
1125 file path=usr/include/sys/ib/ibtl/ibtl_ci_types.h
1126 file path=usr/include/sys/ib/ibtl/ibtl_status.h
1127 file path=usr/include/sys/ib/ibtl/ibtl_types.h
1128 file path=usr/include/sys/ib/ibtl/ibvti.h
1129 file path=usr/include/sys/ib/ibtl/impl/ibtl_util.h
1130 file path=usr/include/sys/ib/mgt/ib_dm_attr.h
1131 file path=usr/include/sys/ib/mgt/ib_mad.h
1132 file path=usr/include/sys/ib/mgt/ibmf/ibmf.h
1133 file path=usr/include/sys/ib/mgt/ibmf/ibmf_msg.h
1134 file path=usr/include/sys/ib/mgt/ibmf/ibmf_saa.h
1135 file path=usr/include/sys/ib/mgt/ibmf/ibmf_utils.h
1136 file path=usr/include/sys/ib/mgt/sa_recs.h
1137 file path=usr/include/sys/ib/mgt/sm_attr.h
1138 file path=usr/include/sys/ibpart.h
1139 file path=usr/include/sys/id32.h
1140 file path=usr/include/sys/id_space.h
1141 file path=usr/include/sys/idmap.h
1142 file path=usr/include/sys/inline.h
1143 file path=usr/include/sys/instance.h
1144 file path=usr/include/sys/int_const.h
1145 file path=usr/include/sys/int_fmtio.h
1146 file path=usr/include/sys/int_limits.h
1147 file path=usr/include/sys/int_types.h
1148 file path=usr/include/sys/inttypes.h
1149 file path=usr/include/sys/ioccom.h
1150 file path=usr/include/sys/ioctl.h
1151 $(i386_ONLY)file path=usr/include/sys/iomulib.h
1152 file path=usr/include/sys/ipc.h
1153 file path=usr/include/sys/ipc_impl.h
1154 file path=usr/include/sys/ipc_rctl.h
1155 file path=usr/include/sys/isa_defs.h
1156 file path=usr/include/sys/iso/signal_iso.h
1157 file path=usr/include/sys/jioctl.h
1158 file path=usr/include/sys/kbd.h
1159 file path=usr/include/sys/kbdreg.h
1160 file path=usr/include/sys/kbio.h
1161 file path=usr/include/sys/kcpc.h
1162 file path=usr/include/sys/kd.h
1163 file path=usr/include/sys/kdi.h
1164 file path=usr/include/sys/kdi_impl.h
1165 file path=usr/include/sys/kdi_machimpl.h
1166 $(i386_ONLY)file path=usr/include/sys/kdi_regs.h
1167 file path=usr/include/sys/kiconv.h
1168 file path=usr/include/sys/kidmap.h
1169 file path=usr/include/sys/klpd.h
1170 file path=usr/include/sys/klwp.h
1171 file path=usr/include/sys/kmem.h
1172 file path=usr/include/sys/kmem_impl.h
1173 file path=usr/include/sys/kobj.h
1174 file path=usr/include/sys/kobj_impl.h
1175 file path=usr/include/sys/ksocket.h
1176 file path=usr/include/sys/kstat.h
1177 file path=usr/include/sys/kstr.h
1178 file path=usr/include/sys/ksyms.h
1179 file path=usr/include/sys/ksynch.h
1180 file path=usr/include/sys/lc_core.h
1181 file path=usr/include/sys/ldterm.h
1182 file path=usr/include/sys/lgrp.h
1183 file path=usr/include/sys/lgrp_user.h
```

```
1184 file path=usr/include/sys/link.h
1185 file path=usr/include/sys/list.h
1186 file path=usr/include/sys/list_impl.h
1187 file path=usr/include/sys/lld.h
1188 file path=usr/include/sys/loadavg.h
1189 file path=usr/include/sys/localedef.h
1190 file path=usr/include/sys/lock.h
1191 file path=usr/include/sys/lockfs.h
1192 file path=usr/include/sys/lofi.h
1193 file path=usr/include/sys/log.h
1194 file path=usr/include/sys/logindmux.h
1195 file path=usr/include/sys/lvm/md_basic.h
1196 file path=usr/include/sys/lvm/md_convert.h
1197 file path=usr/include/sys/lvm/md_crc.h
1198 file path=usr/include/sys/lvm/md_hotspares.h
1199 file path=usr/include/sys/lvm/md_mddb.h
1200 file path=usr/include/sys/lvm/md_mdiox.h
1201 file path=usr/include/sys/lvm/md_mhdx.h
1202 file path=usr/include/sys/lvm/md_mirror.h
1203 file path=usr/include/sys/lvm/md_mirror_shared.h
1204 file path=usr/include/sys/lvm/md_names.h
1205 file path=usr/include/sys/lvm/md_notify.h
1206 file path=usr/include/sys/lvm/md_raid.h
1207 file path=usr/include/sys/lvm/md_rename.h
1208 file path=usr/include/sys/lvm/md_sp.h
1209 file path=usr/include/sys/lvm/md_stripe.h
1210 file path=usr/include/sys/lvm/md_trans.h
1211 file path=usr/include/sys/lvm/mdio.h
1212 file path=usr/include/sys/lvm/mdmed.h
1213 file path=usr/include/sys/lvm/mdmn_commd.h
1214 file path=usr/include/sys/lvm/mdvar.h
1215 file path=usr/include/sys/lwp.h
1216 file path=usr/include/sys/lwp_timer_impl.h
1217 file path=usr/include/sys/lwp_upimutex_impl.h
1218 file path=usr/include/sys/mac.h
1219 file path=usr/include/sys/mac_ether.h
1220 file path=usr/include/sys/mac_flow.h
1221 file path=usr/include/sys/mac_provider.h
1222 file path=usr/include/sys/machelf.h
1223 file path=usr/include/sys/machlock.h
1224 file path=usr/include/sys/machsig.h
1225 file path=usr/include/sys/machtypes.h
1226 file path=usr/include/sys/map.h
1227 $(i386_ONLY)file path=usr/include/sys/mc.h
1228 $(i386_ONLY)file path=usr/include/sys/mc_amd.h
1229 $(i386_ONLY)file path=usr/include/sys/mc_intel.h
1230 $(i386_ONLY)file path=usr/include/sys/mca_amd.h
1231 $(i386_ONLY)file path=usr/include/sys/mca_x86.h
1232 file path=usr/include/sys/md4.h
1233 file path=usr/include/sys/md5.h
1234 file path=usr/include/sys/md5_consts.h
1235 file path=usr/include/sys/mdi_impldefs.h
1236 file path=usr/include/sys/mem.h
1237 file path=usr/include/sys/mem_config.h
1238 file path=usr/include/sys/memlist.h
1239 file path=usr/include/sys/mhd.h
1240 file path=usr/include/sys/mii.h
1241 file path=usr/include/sys/miiregs.h
1242 file path=usr/include/sys/mkdev.h
1243 file path=usr/include/sys/mman.h
1244 file path=usr/include/sys/mmapiobj.h
1245 file path=usr/include/sys/mntent.h
1246 file path=usr/include/sys/mntio.h
1247 file path=usr/include/sys/mnttab.h
1248 file path=usr/include/sys/modctl.h
1249 file path=usr/include/sys/mode.h
```

```
1250 file path=usr/include/sys/model.h
1251 file path=usr/include/sys/modhash.h
1252 file path=usr/include/sys/modhash_impl.h
1253 file path=usr/include/sys/mount.h
1254 file path=usr/include/sys/mouse.h
1255 file path=usr/include/sys/msacct.h
1256 file path=usr/include/sys/msg.h
1257 file path=usr/include/sys/msg_impl.h
1258 file path=usr/include/sys/msio.h
1259 file path=usr/include/sys/msreg.h
1260 file path=usr/include/sys/mtio.h
1261 file path=usr/include/sys/multidata.h
1262 file path=usr/include/sys/mutex.h
1263 $(i386_ONLY)file path=usr/include/sys/mutex_impl.h
1264 file path=usr/include/sys/nbmlock.h
1265 file path=usr/include/sys/ndi_impldefs.h
1266 file path=usr/include/sys/ndifm.h
1267 file path=usr/include/sys/netconfig.h
1268 file path=usr/include/sys/neti.h
1269 file path=usr/include/sys/netstack.h
1270 file path=usr/include/sys/nexusdefs.h
1271 file path=usr/include/sys/note.h
1272 file path=usr/include/sys/nvpair.h
1273 file path=usr/include/sys/nvpair_impl.h
1274 file path=usr/include/sys/objfs.h
1275 file path=usr/include/sys/objfs_impl.h
1276 file path=usr/include/sys/obpdefs.h
1277 file path=usr/include/sys/old_procfs.h
1278 file path=usr/include/sys/open.h
1279 file path=usr/include/sys/openpromio.h
1280 file path=usr/include/sys/panic.h
1281 file path=usr/include/sys/param.h
1282 file path=usr/include/sys/pathconf.h
1283 file path=usr/include/sys/pathname.h
1284 file path=usr/include/sys/pattr.h
1285 file path=usr/include/sys/pbio.h
1286 file path=usr/include/sys/pcb.h
1287 file path=usr/include/sys/pccard.h
1288 file path=usr/include/sys/pci.h
1289 $(i386_ONLY)file path=usr/include/sys/pcic_reg.h
1290 $(i386_ONLY)file path=usr/include/sys/pcic_var.h
1291 file path=usr/include/sys/pcie.h
1292 file path=usr/include/sys/pcmcia.h
1293 file path=usr/include/sys/pcmcia/pcata.h
1294 file path=usr/include/sys/pcmcia/pcser_conf.h
1295 file path=usr/include/sys/pcmcia/pcser_io.h
1296 file path=usr/include/sys/pcmcia/pcser_manuspec.h
1297 file path=usr/include/sys/pcmcia/pcser_reg.h
1298 file path=usr/include/sys/pcmcia/pcser_var.h
1299 file path=usr/include/sys/pctypes.h
1300 file path=usr/include/sys/pfmod.h
1301 file path=usr/include/sys/pg.h
1302 file path=usr/include/sys/pghw.h
1303 file path=usr/include/sys/phymem.h
1304 $(i386_ONLY)file path=usr/include/sys/pic.h
1305 $(i386_ONLY)file path=usr/include/sys/pit.h
1306 file path=usr/include/sys/pkg_hash.h
1307 file path=usr/include/sys/pm.h
1308 $(i386_ONLY)file path=usr/include/sys/pmem.h
1309 file path=usr/include/sys/policy.h
1310 file path=usr/include/sys/poll.h
1311 file path=usr/include/sys/poll_impl.h
1312 file path=usr/include/sys/pool.h
1313 file path=usr/include/sys/pool_impl.h
1314 file path=usr/include/sys/pool_pset.h
1315 file path=usr/include/sys/port.h
```

```
1316 file path=usr/include/sys/port_impl.h
1317 file path=usr/include/sys/port_kernel.h
1318 file path=usr/include/sys/ppmio.h
1319 file path=usr/include/sys/priocntl.h
1320 file path=usr/include/sys/priv.h
1321 file path=usr/include/sys/priv_const.h
1322 file path=usr/include/sys/priv_impl.h
1323 file path=usr/include/sys/priv_names.h
1324 $(i386_ONLY)file path=usr/include/sys/privregs.h
1325 $(i386_ONLY)file path=usr/include/sys/privregs.h
1326 file path=usr/include/sys/prnio.h
1327 file path=usr/include/sys/proc.h
1328 file path=usr/include/sys/proc/prdata.h
1329 file path=usr/include/sys/processor.h
1330 file path=usr/include/sys/procfs.h
1331 file path=usr/include/sys/procfs_isa.h
1332 file path=usr/include/sys/procset.h
1333 file path=usr/include/sys/project.h
1334 $(i386_ONLY)file path=usr/include/sys/prom_emul.h
1335 $(i386_ONLY)file path=usr/include/sys/prom_isa.h
1336 $(i386_ONLY)file path=usr/include/sys/prom_plat.h
1337 file path=usr/include/sys/promif.h
1338 file path=usr/include/sys/promimpl.h
1339 file path=usr/include/sys/protosw.h
1340 file path=usr/include/sys/prsystem.h
1341 file path=usr/include/sys/pset.h
1342 file path=usr/include/sys/psw.h
1343 $(i386_ONLY)file path=usr/include/sys/pte.h
1344 file path=usr/include/sys/ptem.h
1345 file path=usr/include/sys/ptms.h
1346 file path=usr/include/sys/ptyvar.h
1347 file path=usr/include/sys/queue.h
1348 file path=usr/include/sys/raidioctl.h
1349 file path=usr/include/sys/ramdisk.h
1350 file path=usr/include/sys/random.h
1351 file path=usr/include/sys/rctl.h
1352 file path=usr/include/sys/rctl_impl.h
1353 file path=usr/include/sys/rds.h
1354 file path=usr/include/sys/reboot.h
1355 file path=usr/include/sys/refstr.h
1356 file path=usr/include/sys/refstr_impl.h
1357 file path=usr/include/sys/reg.h
1358 file path=usr/include/sys/regset.h
1359 file path=usr/include/sys/resource.h
1360 file path=usr/include/sys/rliocntl.h
1361 file path=usr/include/sys/rsm/rsm.h
1362 file path=usr/include/sys/rsm/rsm_common.h
1363 file path=usr/include/sys/rsm/rsmapi_common.h
1364 file path=usr/include/sys/rsm/rsmka_path_int.h
1365 file path=usr/include/sys/rsm/rsmmdi.h
1366 file path=usr/include/sys/rsm/rsmapi.h
1367 file path=usr/include/sys/rsm/rsmapi_driver.h
1368 file path=usr/include/sys/rt.h
1369 $(i386_ONLY)file path=usr/include/sys/rtc.h
1370 file path=usr/include/sys/rtpriocntl.h
1371 file path=usr/include/sys/rwlock.h
1372 file path=usr/include/sys/rwlock_impl.h
1373 file path=usr/include/sys/rwstlock.h
1374 file path=usr/include/sys/sad.h
1375 $(i386_ONLY)file path=usr/include/sys/sata/sata_defs.h
1376 $(i386_ONLY)file path=usr/include/sys/sata/sata_hba.h
1377 file path=usr/include/sys/schedctl.h
1378 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/ifpio.h
1379 file path=usr/include/sys/scsi/adapters/scsi_vhci.h
1380 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/sfvar.h
1381 file path=usr/include/sys/scsi/conf/autoconf.h
```

```
1382 file path=usr/include/sys/scsi/conf/device.h
1383 file path=usr/include/sys/scsi/generic/commands.h
1384 file path=usr/include/sys/scsi/generic/dad_mode.h
1385 file path=usr/include/sys/scsi/generic/inquiry.h
1386 file path=usr/include/sys/scsi/generic/message.h
1387 file path=usr/include/sys/scsi/generic/mode.h
1388 file path=usr/include/sys/scsi/generic/persist.h
1389 file path=usr/include/sys/scsi/generic/sense.h
1390 file path=usr/include/sys/scsi/generic/sff_frames.h
1391 file path=usr/include/sys/scsi/generic/smp_frames.h
1392 file path=usr/include/sys/scsi/generic/status.h
1393 file path=usr/include/sys/scsi/impl/commands.h
1394 file path=usr/include/sys/scsi/impl/inquiry.h
1395 file path=usr/include/sys/scsi/impl/mode.h
1396 file path=usr/include/sys/scsi/impl/scsi_reset_notify.h
1397 file path=usr/include/sys/scsi/impl/scsi_sas.h
1398 file path=usr/include/sys/scsi/impl/sense.h
1399 file path=usr/include/sys/scsi/impl/services.h
1400 file path=usr/include/sys/scsi/impl/smp_transport.h
1401 file path=usr/include/sys/scsi/impl/spc3_types.h
1402 file path=usr/include/sys/scsi/impl/status.h
1403 file path=usr/include/sys/scsi/impl/transport.h
1404 file path=usr/include/sys/scsi/impl/types.h
1405 file path=usr/include/sys/scsi/impl/uscsi.h
1406 file path=usr/include/sys/scsi/impl/usmp.h
1407 file path=usr/include/sys/scsi/scsi.h
1408 file path=usr/include/sys/scsi/scsi_address.h
1409 file path=usr/include/sys/scsi/scsi_ctl.h
1410 file path=usr/include/sys/scsi/scsi_fm.h
1411 file path=usr/include/sys/scsi/scsi_params.h
1412 file path=usr/include/sys/scsi/scsi_pkt.h
1413 file path=usr/include/sys/scsi/scsi_resource.h
1414 file path=usr/include/sys/scsi/scsi_types.h
1415 file path=usr/include/sys/scsi/scsi_watch.h
1416 file path=usr/include/sys/scsi/targets/sddef.h
1417 file path=usr/include/sys/scsi/targets/ses.h
1418 file path=usr/include/sys/scsi/targets/sesio.h
1419 file path=usr/include/sys/scsi/targets/sgendef.h
1420 file path=usr/include/sys/scsi/targets/smp.h
1421 $(sparc_ONLY)file path=usr/include/sys/scsi/targets/ssddef.h
1422 file path=usr/include/sys/scsi/targets/stddef.h
1423 $(i386_ONLY)file path=usr/include/sys/segment.h
1424 $(i386_ONLY)file path=usr/include/sys/segments.h
1425 file path=usr/include/sys/select.h
1426 file path=usr/include/sys/sem.h
1427 file path=usr/include/sys/sem_impl.h
1428 file path=usr/include/sys/semaphore.h
1429 file path=usr/include/sys/semaphore.h
1430 file path=usr/include/sys/sendfile.h
1431 $(sparc_ONLY)file path=usr/include/sys/ser_async.h
1432 file path=usr/include/sys/ser_sync.h
1433 $(sparc_ONLY)file path=usr/include/sys/ser_zscc.h
1434 file path=usr/include/sys/serializer.h
1435 file path=usr/include/sys/session.h
1436 file path=usr/include/sys/sha1.h
1437 file path=usr/include/sys/sha2.h
1438 file path=usr/include/sys/share.h
1439 file path=usr/include/sys/shm.h
1440 file path=usr/include/sys/shm_impl.h
1441 file path=usr/include/sys/sid.h
1442 file path=usr/include/sys/siginfo.h
1443 file path=usr/include/sys/signal.h
1444 file path=usr/include/sys/sleepq.h
1445 file path=usr/include/sys/smbios.h
1446 file path=usr/include/sys/smbios_impl.h
1447 file path=usr/include/sys/smedia.h
```

```
1448 file path=usr/include/sys/subject.h
1449 $(sparc_ONLY)file path=usr/include/sys/socal_cq_defs.h
1450 $(sparc_ONLY)file path=usr/include/sys/socalio.h
1451 $(sparc_ONLY)file path=usr/include/sys/socalmap.h
1452 $(sparc_ONLY)file path=usr/include/sys/socalreg.h
1453 $(sparc_ONLY)file path=usr/include/sys/socalvar.h
1454 file path=usr/include/sys/socket.h
1455 file path=usr/include/sys/socket_impl.h
1456 file path=usr/include/sys/socket_proto.h
1457 file path=usr/include/sys/socketvar.h
1458 file path=usr/include/sys/sockio.h
1459 file path=usr/include/sys/spl.h
1460 file path=usr/include/sys/squeue.h
1461 file path=usr/include/sys/squeue_impl.h
1462 file path=usr/include/sys/sservice.h
1463 file path=usr/include/sys/stack.h
1464 file path=usr/include/sys/stat.h
1465 file path=usr/include/sys/stat_impl.h
1466 file path=usr/include/sys/statfs.h
1467 file path=usr/include/sys/statvfs.h
1468 file path=usr/include/sys/stdbool.h
1469 file path=usr/include/sys/stdint.h
1470 file path=usr/include/sys/stermio.h
1471 file path=usr/include/sys/stream.h
1472 file path=usr/include/sys/strft.h
1473 file path=usr/include/sys/strlog.h
1474 file path=usr/include/sys/strmdep.h
1475 file path=usr/include/sys/stropts.h
1476 file path=usr/include/sys/strredir.h
1477 file path=usr/include/sys/strstat.h
1478 file path=usr/include/sys/strsubr.h
1479 file path=usr/include/sys/strsun.h
1480 file path=usr/include/sys/strtty.h
1481 file path=usr/include/sys/sunddi.h
1482 file path=usr/include/sys/sunldi.h
1483 file path=usr/include/sys/sunldi_impl.h
1484 file path=usr/include/sys/sunmdi.h
1485 file path=usr/include/sys/sunndi.h
1486 file path=usr/include/sys/sunpm.h
1487 file path=usr/include/sys/suntpi.h
1488 file path=usr/include/sys/suntty.h
1489 file path=usr/include/sys/swap.h
1490 file path=usr/include/sys/synch.h
1491 file path=usr/include/sys/syscall.h
1492 file path=usr/include/sys/sysconf.h
1493 file path=usr/include/sys/sysconfig.h
1494 file path=usr/include/sys/sysconfig_impl.h
1495 file path=usr/include/sys/sysdc.h
1496 file path=usr/include/sys/sysdc_impl.h
1497 file path=usr/include/sys/sysevent.h
1498 file path=usr/include/sys/sysevent/ap_driver.h
1499 file path=usr/include/sys/sysevent/dev.h
1500 file path=usr/include/sys/sysevent/domain.h
1501 file path=usr/include/sys/sysevent/dr.h
1502 file path=usr/include/sys/sysevent/env.h
1503 file path=usr/include/sys/sysevent/eventdefs.h
1504 file path=usr/include/sys/sysevent/imp.h
1505 file path=usr/include/sys/sysevent/pwrctl.h
1506 file path=usr/include/sys/sysevent/svm.h
1507 file path=usr/include/sys/sysevent/vrrp.h
1508 file path=usr/include/sys/sysevent_impl.h
1509 $(i386_ONLY)file path=usr/include/sys/sysi86.h
1510 file path=usr/include/sys/sysinfo.h
1511 file path=usr/include/sys/syslog.h
1512 file path=usr/include/sys/sysmacros.h
1513 file path=usr/include/sys/systeminfo.h
```

```
1514 file path=usr/include/sys/systm.h
1515 file path=usr/include/sys/t_kuser.h
1516 file path=usr/include/sys/t_lock.h
1517 file path=usr/include/sys/task.h
1518 file path=usr/include/sys/taskq.h
1519 file path=usr/include/sys/taskq_impl.h
1520 file path=usr/include/sys/telioctl.h
1521 file path=usr/include/sys/termio.h
1522 file path=usr/include/sys/termios.h
1523 file path=usr/include/sys/termiox.h
1524 file path=usr/include/sys/thread.h
1525 file path=usr/include/sys/ticlts.h
1526 file path=usr/include/sys/ticots.h
1527 file path=usr/include/sys/ticotsord.h
1528 file path=usr/include/sys/tihdr.h
1529 file path=usr/include/sys/time.h
1530 file path=usr/include/sys/time_impl.h
1531 file path=usr/include/sys/time_std_impl.h
1532 file path=usr/include/sys/timeb.h
1533 file path=usr/include/sys/timer.h
1534 file path=usr/include/sys/times.h
1535 file path=usr/include/sys/timex.h
1536 file path=usr/include/sys/timod.h
1537 file path=usr/include/sys/tirdwr.h
1538 file path=usr/include/sys/tiuser.h
1539 file path=usr/include/sys/tl.h
1540 file path=usr/include/sys/tnf.h
1541 file path=usr/include/sys/tnf_com.h
1542 file path=usr/include/sys/tnf_probe.h
1543 file path=usr/include/sys/tnf_writer.h
1544 file path=usr/include/sys/todio.h
1545 file path=usr/include/sys/tpicommon.h
1546 file path=usr/include/sys/trap.h
1547 $(i386_ONLY)file path=usr/include/sys/traptrace.h
1548 file path=usr/include/sys/ts.h
1549 file path=usr/include/sys/tsol/label.h
1550 file path=usr/include/sys/tsol/label_macro.h
1551 file path=usr/include/sys/tsol/priv.h
1552 file path=usr/include/sys/tsol/tndb.h
1553 file path=usr/include/sys/tsol/tsyscall.h
1554 file path=usr/include/sys/tspricntl.h
1555 $(i386_ONLY)file path=usr/include/sys/tss.h
1556 file path=usr/include/sys/ttcompat.h
1557 file path=usr/include/sys/ttold.h
1558 file path=usr/include/sys/tty.h
1559 file path=usr/include/sys/ttychars.h
1560 file path=usr/include/sys/ttydev.h
1561 $(sparc_ONLY)file path=usr/include/sys/ttymux.h
1562 $(sparc_ONLY)file path=usr/include/sys/ttymuxuser.h
1563 file path=usr/include/sys/tuneable.h
1564 file path=usr/include/sys/turnstile.h
1565 file path=usr/include/sys/types.h
1566 file path=usr/include/sys/types32.h
1567 file path=usr/include/sys/tzfile.h
1568 file path=usr/include/sys/u8_textprep.h
1569 file path=usr/include/sys/uadmin.h
1570 $(i386_ONLY)file path=usr/include/sys/ucode.h
1571 file path=usr/include/sys/ucontext.h
1572 file path=usr/include/sys/uio.h
1573 file path=usr/include/sys/ulimit.h
1574 file path=usr/include/sys/un.h
1575 file path=usr/include/sys/unistd.h
1576 file path=usr/include/sys/user.h
1577 file path=usr/include/sys/ustat.h
1578 file path=usr/include/sys/utime.h
1579 file path=usr/include/sys/utrap.h
```

```
1580 file path=usr/include/sys/utsname.h
1581 file path=usr/include/sys/utssys.h
1582 file path=usr/include/sys/uuid.h
1583 file path=usr/include/sys/va_impl.h
1584 file path=usr/include/sys/va_list.h
1585 file path=usr/include/sys/var.h
1586 file path=usr/include/sys/varargs.h
1587 file path=usr/include/sys/vfs.h
1588 file path=usr/include/sys/vfs_opreg.h
1589 file path=usr/include/sys/vfstab.h
1590 file path=usr/include/sys/videodev2.h
1591 file path=usr/include/sys/visual_io.h
1592 file path=usr/include/sys/vm.h
1593 file path=usr/include/sys/vm_usage.h
1594 file path=usr/include/sys/vmem.h
1595 file path=usr/include/sys/vmem_impl.h
1596 file path=usr/include/sys/vmem_impl_user.h
1597 file path=usr/include/sys/vmparam.h
1598 file path=usr/include/sys/vmsystm.h
1599 file path=usr/include/sys/vnode.h
1600 file path=usr/include/sys/vt.h
1601 file path=usr/include/sys/vtdaemon.h
1602 file path=usr/include/sys/vtoc.h
1603 file path=usr/include/sys/vtrace.h
1604 file path=usr/include/sys/vuid_event.h
1605 file path=usr/include/sys/vuid_queue.h
1606 file path=usr/include/sys/vuid_state.h
1607 file path=usr/include/sys/vuid_store.h
1608 file path=usr/include/sys/vuid_wheel.h
1609 file path=usr/include/sys/wait.h
1610 file path=usr/include/sys/waitq.h
1611 file path=usr/include/sys/watchpoint.h
1612 $(i386_ONLY)file path=usr/include/sys/x86_archext.h
1613 $(i386_ONLY)file path=usr/include/sys/xen_errno.h
1614 file path=usr/include/sys/xti_inet.h
1615 file path=usr/include/sys/xti_osi.h
1616 file path=usr/include/sys/xti_xtiopt.h
1617 file path=usr/include/sys/zcons.h
1618 file path=usr/include/sys/zmod.h
1619 file path=usr/include/sys/zone.h
1620 $(sparc_ONLY)file path=usr/include/sys/zsdev.h
1621 file path=usr/include/sys/sextits.h
1622 file path=usr/include/sys/slog.h
1623 file path=usr/include/tar.h
1624 file path=usr/include/tcpd.h
1625 file path=usr/include/term.h
1626 file path=usr/include/termcap.h
1627 file path=usr/include/termio.h
1628 file path=usr/include/termios.h
1629 file path=usr/include/thread.h
1630 file path=usr/include/thread_db.h
1631 file path=usr/include/time.h
1632 file path=usr/include/tiuser.h
1633 file path=usr/include/tsol/label.h
1634 file path=usr/include/tzfile.h
1635 file path=usr/include/ucontext.h
1636 file path=usr/include/ucred.h
1637 file path=usr/include/uid_stp.h
1638 file path=usr/include/ulimit.h
1639 file path=usr/include/umem.h
1640 file path=usr/include/umem_impl.h
1641 file path=usr/include/unctrl.h
1642 file path=usr/include/unistd.h
1643 file path=usr/include/user_attr.h
1644 file path=usr/include/userdefs.h
1645 file path=usr/include/ustat.h
```

```
1646 file path=usr/include/utility.h
1647 file path=usr/include/utime.h
1648 file path=usr/include/utmp.h
1649 file path=usr/include/utmpx.h
1650 file path=usr/include/uuid/uuid.h
1651 $(sparc_ONLY)file path=usr/include/v7/sys/machpcb.h
1652 $(sparc_ONLY)file path=usr/include/v7/sys/machtrap.h
1653 $(sparc_ONLY)file path=usr/include/v7/sys/mutex_impl.h
1654 $(sparc_ONLY)file path=usr/include/v7/sys/privregs.h
1655 $(sparc_ONLY)file path=usr/include/v7/sys/prom_isa.h
1656 $(sparc_ONLY)file path=usr/include/v7/sys/psr.h
1657 $(sparc_ONLY)file path=usr/include/v7/sys/traptrace.h
1658 $(sparc_ONLY)file path=usr/include/v9/sys/asi.h
1659 $(sparc_ONLY)file path=usr/include/v9/sys/machpcb.h
1660 $(sparc_ONLY)file path=usr/include/v9/sys/machtrap.h
1661 $(sparc_ONLY)file path=usr/include/v9/sys/membar.h
1662 $(sparc_ONLY)file path=usr/include/v9/sys/mutex_impl.h
1663 $(sparc_ONLY)file path=usr/include/v9/sys/privregs.h
1664 $(sparc_ONLY)file path=usr/include/v9/sys/prom_isa.h
1665 $(sparc_ONLY)file path=usr/include/v9/sys/psr_compat.h
1666 $(sparc_ONLY)file path=usr/include/v9/sys/vis_simulator.h
1667 file path=usr/include/valtools.h
1668 file path=usr/include/values.h
1669 file path=usr/include/varargs.h
1670 file path=usr/include/vm/anon.h
1671 file path=usr/include/vm/as.h
1672 file path=usr/include/vm/faultcode.h
1673 file path=usr/include/vm/hat.h
1674 file path=usr/include/vm/kpm.h
1675 file path=usr/include/vm/page.h
1676 file path=usr/include/vm/pvn.h
1677 file path=usr/include/vm/rm.h
1678 file path=usr/include/vm/seg.h
1679 file path=usr/include/vm/seg_dev.h
1680 file path=usr/include/vm/seg_enum.h
1681 file path=usr/include/vm/seg_kmem.h
1682 file path=usr/include/vm/seg_kp.h
1683 file path=usr/include/vm/seg_kpm.h
1684 file path=usr/include/vm/seg_map.h
1685 file path=usr/include/vm/seg_spt.h
1686 file path=usr/include/vm/seg_vn.h
1687 file path=usr/include/vm/vpage.h
1688 file path=usr/include/vm/vpm.h
1689 file path=usr/include/volmgt.h
1690 file path=usr/include/wait.h
1691 file path=usr/include/wchar.h
1692 file path=usr/include/wchar_impl.h
1693 file path=usr/include/wctype.h
1694 file path=usr/include/widec.h
1695 file path=usr/include/wordexp.h
1696 file path=usr/include/xti.h
1697 file path=usr/include/xti_inet.h
1698 file path=usr/include/zone.h
1699 file path=usr/include/zonestat.h
1700 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/acpidev.h
1701 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/amd_iommu.h
1702 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/asm_misc.h
1703 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/clock.h
1704 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/cram.h
1705 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/ddi_subrdefs.h
1706 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/debug_info.h
1707 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/fastboot.h
1708 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/mach_mmu.h
1709 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machclock.h
1710 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machcpuvar.h
1711 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machparam.h
```



```

1712 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machprivregs.h
1713 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machsystem.h
1714 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machthread.h
1715 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/memnode.h
1716 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/pc_mmu.h
1717 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm.h
1718 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_defs.h
1719 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_modctl.h
1720 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_types.h
1721 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/rm_platter.h
1722 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/sbd_ioctl.h
1723 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/smp_impldefs.h
1724 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/vm_machparam.h
1725 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/x_call.h
1726 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xc_levels.h
1727 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xsvc.h
1728 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_i86.h
1729 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_pte.h
1730 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hment.h
1731 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/htable.h
1732 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/kboot_mmu.h
1733 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/balloon.h
1734 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/machprivregs.h
1735 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xen_mmu.h
1736 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xpv_impl.h
1737 $(i386_ONLY)file path=usr/platform/i86xpv/include/vm/seg_mf.h
1738 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ac.h
1739 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/async.h
1740 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cheetahregs.h
1741 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cherrystone.h
1742 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/clock.h
1743 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cmp.h
1744 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpc_ultra.h
1745 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpr_impl.h
1746 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_impl.h
1747 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_sgnblk_defs.h
1748 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cvc.h
1749 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/daktari.h
1750 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ddi_subrdefs.h
1751 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/dvma.h
1752 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ecc_kstat.h
1753 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/EEPROM.h
1754 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl.h
1755 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_gen.h
1756 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue250.h
1757 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue450.h
1758 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envron.h
1759 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/errclassify.h
1760 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/fhc.h
1761 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/gpio_87317.h
1762 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/hpc3130_events.h
1763 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/hpc3130.h
1764 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/i2c_client.h
1765 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/lm75.h
1766 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/max1617.h
1767 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/pcf8591.h
1768 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c_clients/ssc050.h
1769 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/misc/i2c_svc.h
1770 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/idprom.h
1771 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intr.h
1772 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intreg.h
1773 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iocache.h
1774 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/immu.h
1775 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ivintr.h
1776 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/lom_io.h
1777 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machasi.h

```

```

1778 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machclock.h
1779 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machcpuvar.h
1780 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machparam.h
1781 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machsystem.h
1782 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machthread.h
1783 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mem_cache.h
1784 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memlist_plat.h
1785 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memnode.h
1786 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mmu.h
1787 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/nexusdebug.h
1788 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_hwdesc.h
1789 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_module.h
1790 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_debug.h
1791 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_plat.h
1792 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/pte.h
1793 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sbd_ioctl.h
1794 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scb.h
1795 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scsb_led.h
1796 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/simstat.h
1797 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sptregs.h
1798 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sram.h
1799 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/starfire.h
1800 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sun4asi.h
1801 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysctrl.h
1802 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysoerr.h
1803 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/syiosbus.h
1804 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tod.h
1805 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/todmostek.h
1806 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/trapstat.h
1807 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/traptrace.h
1808 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vis.h
1809 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vm_machparam.h
1810 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/x_call.h
1811 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/xc_impl.h
1812 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/zsmach.h
1813 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/hat_sfmmu.h
1814 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/mach_sfmmu.h
1815 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/clock.h
1816 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cmp.h
1817 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpc_ultra.h
1818 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpu_sgnblk_defs.h
1819 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ddi_subrdefs.h
1820 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
1821 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_snmp.h
1822 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
1823 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/EEPROM.h
1824 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/fcode.h
1825 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hsvc.h
1826 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hypervisor_api.h
1827 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/idprom.h
1828 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intr.h
1829 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intreg.h
1830 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ivintr.h
1831 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machasi.h
1832 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machclock.h
1833 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machcpuvar.h
1834 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machintreg.h
1835 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machparam.h
1836 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machsystem.h
1837 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machthread.h
1838 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memlist_plat.h
1839 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memnode.h
1840 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/mmu.h
1841 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/nexusdebug.h
1842 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagaraasi.h
1843 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagararegs.h

```

```
1844 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ntwtdt.h
1845 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pri.h
1846 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_debug.h
1847 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_plat.h
1848 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pte.h
1849 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/qcn.h
1850 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/scb.h
1851 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/soft_state.h
1852 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/sun4asi.h
1853 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/tod.h
1854 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/trapstat.h
1855 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/traptrace.h
1856 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vis.h
1857 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vm_machparam.h
1858 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/x_call.h
1859 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/xc_impl.h
1860 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/zsmach.h
1861 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/hat_sfmmu.h
1862 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/mach_sfmmu.h
1863 file path=usr/share/man/man3head/acct.3head
1864 file path=usr/share/man/man3head/acct.h.3head
1865 file path=usr/share/man/man3head/aio.3head
1866 file path=usr/share/man/man3head/aio.h.3head
1867 file path=usr/share/man/man3head/ar.3head
1868 file path=usr/share/man/man3head/ar.h.3head
1869 file path=usr/share/man/man3head/archives.3head
1870 file path=usr/share/man/man3head/archives.h.3head
1871 file path=usr/share/man/man3head/assert.3head
1872 file path=usr/share/man/man3head/assert.h.3head
1873 file path=usr/share/man/man3head/complex.3head
1874 file path=usr/share/man/man3head/complex.h.3head
1875 file path=usr/share/man/man3head/cpio.3head
1876 file path=usr/share/man/man3head/cpio.h.3head
1877 file path=usr/share/man/man3head/dirent.3head
1878 file path=usr/share/man/man3head/dirent.h.3head
1879 file path=usr/share/man/man3head/errno.3head
1880 file path=usr/share/man/man3head/errno.h.3head
1881 file path=usr/share/man/man3head/fcntl.3head
1882 file path=usr/share/man/man3head/fcntl.h.3head
1883 file path=usr/share/man/man3head/fenv.3head
1884 file path=usr/share/man/man3head/fenv.h.3head
1885 file path=usr/share/man/man3head/float.3head
1886 file path=usr/share/man/man3head/float.h.3head
1887 file path=usr/share/man/man3head/floatingpoint.3head
1888 file path=usr/share/man/man3head/floatingpoint.h.3head
1889 file path=usr/share/man/man3head/fmtmsg.3head
1890 file path=usr/share/man/man3head/fmtmsg.h.3head
1891 file path=usr/share/man/man3head/fnmatch.3head
1892 file path=usr/share/man/man3head/fnmatch.h.3head
1893 file path=usr/share/man/man3head/ftw.3head
1894 file path=usr/share/man/man3head/ftw.h.3head
1895 file path=usr/share/man/man3head/glob.3head
1896 file path=usr/share/man/man3head/glob.h.3head
1897 file path=usr/share/man/man3head/grp.3head
1898 file path=usr/share/man/man3head/grp.h.3head
1899 file path=usr/share/man/man3head/iconv.3head
1900 file path=usr/share/man/man3head/iconv.h.3head
1901 file path=usr/share/man/man3head/if.3head
1902 file path=usr/share/man/man3head/if.h.3head
1903 file path=usr/share/man/man3head/in.3head
1904 file path=usr/share/man/man3head/in.h.3head
1905 file path=usr/share/man/man3head/inet.3head
1906 file path=usr/share/man/man3head/inet.h.3head
1907 file path=usr/share/man/man3head/inttypes.3head
1908 file path=usr/share/man/man3head/inttypes.h.3head
1909 file path=usr/share/man/man3head/ipc.3head
```

```
1910 file path=usr/share/man/man3head/ipc.h.3head
1911 file path=usr/share/man/man3head/iso646.3head
1912 file path=usr/share/man/man3head/iso646.h.3head
1913 file path=usr/share/man/man3head/langinfo.3head
1914 file path=usr/share/man/man3head/langinfo.h.3head
1915 file path=usr/share/man/man3head/libgen.3head
1916 file path=usr/share/man/man3head/libgen.h.3head
1917 file path=usr/share/man/man3head/libintl.3head
1918 file path=usr/share/man/man3head/libintl.h.3head
1919 file path=usr/share/man/man3head/limits.3head
1920 file path=usr/share/man/man3head/limits.h.3head
1921 file path=usr/share/man/man3head/locale.3head
1922 file path=usr/share/man/man3head/locale.h.3head
1923 file path=usr/share/man/man3head/math.3head
1924 file path=usr/share/man/man3head/math.h.3head
1925 file path=usr/share/man/man3head/mman.3head
1926 file path=usr/share/man/man3head/mman.h.3head
1927 file path=usr/share/man/man3head/monetary.3head
1928 file path=usr/share/man/man3head/monetary.h.3head
1929 file path=usr/share/man/man3head/mqueue.3head
1930 file path=usr/share/man/man3head/mqueue.h.3head
1931 file path=usr/share/man/man3head/msg.3head
1932 file path=usr/share/man/man3head/msg.h.3head
1933 file path=usr/share/man/man3head/ndbm.3head
1934 file path=usr/share/man/man3head/ndbm.h.3head
1935 file path=usr/share/man/man3head/netdb.3head
1936 file path=usr/share/man/man3head/netdb.h.3head
1937 file path=usr/share/man/man3head/nl_types.3head
1938 file path=usr/share/man/man3head/nl_types.h.3head
1939 file path=usr/share/man/man3head/poll.3head
1940 file path=usr/share/man/man3head/poll.h.3head
1941 file path=usr/share/man/man3head/pthread.3head
1942 file path=usr/share/man/man3head/pthread.h.3head
1943 file path=usr/share/man/man3head/pwd.3head
1944 file path=usr/share/man/man3head/pwd.h.3head
1945 file path=usr/share/man/man3head/regex.3head
1946 file path=usr/share/man/man3head/regex.h.3head
1947 file path=usr/share/man/man3head/resource.3head
1948 file path=usr/share/man/man3head/resource.h.3head
1949 file path=usr/share/man/man3head/sched.3head
1950 file path=usr/share/man/man3head/sched.h.3head
1951 file path=usr/share/man/man3head/search.3head
1952 file path=usr/share/man/man3head/search.h.3head
1953 file path=usr/share/man/man3head/select.3head
1954 file path=usr/share/man/man3head/select.h.3head
1955 file path=usr/share/man/man3head/sem.3head
1956 file path=usr/share/man/man3head/sem.h.3head
1957 file path=usr/share/man/man3head/semaphore.3head
1958 file path=usr/share/man/man3head/semaphore.h.3head
1959 file path=usr/share/man/man3head/setjmp.3head
1960 file path=usr/share/man/man3head/setjmp.h.3head
1961 file path=usr/share/man/man3head/shm.3head
1962 file path=usr/share/man/man3head/shm.h.3head
1963 file path=usr/share/man/man3head/siginfo.3head
1964 file path=usr/share/man/man3head/siginfo.h.3head
1965 file path=usr/share/man/man3head/signal.3head
1966 file path=usr/share/man/man3head/signal.h.3head
1967 file path=usr/share/man/man3head/socket.3head
1968 file path=usr/share/man/man3head/socket.h.3head
1969 file path=usr/share/man/man3head/spawn.3head
1970 file path=usr/share/man/man3head/spawn.h.3head
1971 file path=usr/share/man/man3head/stat.3head
1972 file path=usr/share/man/man3head/stat.h.3head
1973 file path=usr/share/man/man3head/statvfs.3head
1974 file path=usr/share/man/man3head/statvfs.h.3head
1975 file path=usr/share/man/man3head/stdbool.3head
```

```
1976 file path=usr/share/man/man3head/stdbool.h.3head
1977 file path=usr/share/man/man3head/stddef.3head
1978 file path=usr/share/man/man3head/stddef.h.3head
1979 file path=usr/share/man/man3head/stdint.3head
1980 file path=usr/share/man/man3head/stdint.h.3head
1981 file path=usr/share/man/man3head/stdio.3head
1982 file path=usr/share/man/man3head/stdio.h.3head
1983 file path=usr/share/man/man3head/stdlib.3head
1984 file path=usr/share/man/man3head/stdlib.h.3head
1985 file path=usr/share/man/man3head/string.3head
1986 file path=usr/share/man/man3head/string.h.3head
1987 file path=usr/share/man/man3head/strings.3head
1988 file path=usr/share/man/man3head/strings.h.3head
1989 file path=usr/share/man/man3head/stropts.3head
1990 file path=usr/share/man/man3head/stropts.h.3head
1991 file path=usr/share/man/man3head/syslog.3head
1992 file path=usr/share/man/man3head/syslog.h.3head
1993 file path=usr/share/man/man3head/tar.3head
1994 file path=usr/share/man/man3head/tar.h.3head
1995 file path=usr/share/man/man3head/tcp.3head
1996 file path=usr/share/man/man3head/tcp.h.3head
1997 file path=usr/share/man/man3head/termios.3head
1998 file path=usr/share/man/man3head/termios.h.3head
1999 file path=usr/share/man/man3head/tgmath.3head
2000 file path=usr/share/man/man3head/tgmath.h.3head
2001 file path=usr/share/man/man3head/time.3head
2002 file path=usr/share/man/man3head/time.h.3head
2003 file path=usr/share/man/man3head/timeb.3head
2004 file path=usr/share/man/man3head/timeb.h.3head
2005 file path=usr/share/man/man3head/times.3head
2006 file path=usr/share/man/man3head/times.h.3head
2007 file path=usr/share/man/man3head/types.3head
2008 file path=usr/share/man/man3head/types.h.3head
2009 file path=usr/share/man/man3head/types32.3head
2010 file path=usr/share/man/man3head/types32.h.3head
2011 file path=usr/share/man/man3head/ucontext.3head
2012 file path=usr/share/man/man3head/ucontext.h.3head
2013 file path=usr/share/man/man3head/uio.3head
2014 file path=usr/share/man/man3head/uio.h.3head
2015 file path=usr/share/man/man3head/ulimit.3head
2016 file path=usr/share/man/man3head/ulimit.h.3head
2017 file path=usr/share/man/man3head/un.3head
2018 file path=usr/share/man/man3head/un.h.3head
2019 file path=usr/share/man/man3head/unistd.3head
2020 file path=usr/share/man/man3head/unistd.h.3head
2021 file path=usr/share/man/man3head/utime.3head
2022 file path=usr/share/man/man3head/utime.h.3head
2023 file path=usr/share/man/man3head/utmpx.3head
2024 file path=usr/share/man/man3head/utmpx.h.3head
2025 file path=usr/share/man/man3head/utsname.3head
2026 file path=usr/share/man/man3head/utsname.h.3head
2027 file path=usr/share/man/man3head/values.3head
2028 file path=usr/share/man/man3head/values.h.3head
2029 file path=usr/share/man/man3head/wait.3head
2030 file path=usr/share/man/man3head/wait.h.3head
2031 file path=usr/share/man/man3head/wchar.3head
2032 file path=usr/share/man/man3head/wchar.h.3head
2033 file path=usr/share/man/man3head/wctype.3head
2034 file path=usr/share/man/man3head/wctype.h.3head
2035 file path=usr/share/man/man3head/wordexp.3head
2036 file path=usr/share/man/man3head/wordexp.h.3head
2037 file path=usr/share/man/man4/note.4
2038 file path=usr/share/man/man5/prof.5
2039 file path=usr/share/man/man7i/cdio.7i
2040 file path=usr/share/man/man7i/dkio.7i
2041 file path=usr/share/man/man7i/fbio.7i
```

```
2042 file path=usr/share/man/man7i/fdio.7i
2043 file path=usr/share/man/man7i/hdio.7i
2044 file path=usr/share/man/man7i/iec61883.7i
2045 file path=usr/share/man/man7i/mhd.7i
2046 file path=usr/share/man/man7i/mtio.7i
2047 file path=usr/share/man/man7i/prnio.7i
2048 file path=usr/share/man/man7i/quotactl.7i
2049 file path=usr/share/man/man7i/sesio.7i
2050 file path=usr/share/man/man7i/sockio.7i
2051 file path=usr/share/man/man7i/streamio.7i
2052 file path=usr/share/man/man7i/termio.7i
2053 file path=usr/share/man/man7i/termiox.7i
2054 file path=usr/share/man/man7i/uscsi.7i
2055 file path=usr/share/man/man7i/visual_io.7i
2056 file path=usr/share/man/man7i/vt.7i
2057 file path=usr/xpg4/include/curses.h
2058 file path=usr/xpg4/include/term.h
2059 file path=usr/xpg4/include/unctrl.h
2060 legacy pkg=SUNWhea \
2061     desc="SunOS C/C++ header files for general development of software" \
2062     name="SunOS Header Files"
2063 license cr_Sun license=cr_Sun
2064 license lic_CDDL license=lic_CDDL
2065 license license_in_headers license=license_in_headers
2066 license usr/src/lib/pkcs11/include/THIRDPARTYLICENSE \
2067     license=usr/src/lib/pkcs11/include/THIRDPARTYLICENSE
2068 link path=usr/include/iso/assert_iso.h target=../assert.h
2069 link path=usr/include/iso/errno_iso.h target=../errno.h
2070 link path=usr/include/iso/float_iso.h target=../float.h
2071 link path=usr/include/iso/iso646_iso.h target=../iso646.h
2072 $(sparc_ONLY)link path=usr/platform/SUNW,A70/include target=../sun4u/include
2073 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T12/include \
2074     target=../sun4u/include
2075 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T4/include \
2076     target=../sun4u/include
2077 $(sparc_ONLY)link path=usr/platform/SUNW,SPARC-Enterprise/include \
2078     target=../sun4u/include
2079 $(sparc_ONLY)link path=usr/platform/SUNW,Serverblade1/include \
2080     target=../sun4u/include
2081 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-100/include \
2082     target=../sun4u/include
2083 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1000/include \
2084     target=../sun4u/include
2085 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1500/include \
2086     target=../sun4u/include
2087 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-2500/include \
2088     target=../sun4u/include
2089 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-15000/include \
2090     target=../sun4u/include
2091 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-280R/include \
2092     target=../sun4u/include
2093 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-480R/include \
2094     target=../sun4u/include
2095 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-880/include \
2096     target=../sun4u/include
2097 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V215/include \
2098     target=../sun4u/include
2099 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V240/include \
2100     target=../sun4u/include
2101 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V250/include \
2102     target=../sun4u/include
2103 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V440/include \
2104     target=../sun4u/include
2105 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V445/include \
2106     target=../sun4u/include
2107 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V490/include \
```

```
2108     target=../sun4u/include
2109 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V890/include \
2110     target=../sun4u/include
2111 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire/include \
2112     target=../sun4u/include
2113 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-2/include \
2114     target=../sun4u/include
2115 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-250/include \
2116     target=../sun4u/include
2117 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-4/include \
2118     target=../sun4u/include
2119 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise-10000/include \
2120     target=../sun4u/include
2121 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise/include \
2122     target=../sun4u/include
2123 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40/include \
2124     target=../sun4u/include
2125 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60/include \
2126     target=../sun4u/include
2127 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-III-Netract/include \
2128     target=../sun4u/include
2129 $(i386_ONLY)link path=usr/share/src/uts/i86pc/sys \
2130     target=../../../../platform/i86pc/include/sys
2131 $(i386_ONLY)link path=usr/share/src/uts/i86pc/vm \
2132     target=../../../../platform/i86pc/include/vm
2133 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/sys \
2134     target=../../../../platform/i86xpv/include/sys
2135 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/vm \
2136     target=../../../../platform/i86xpv/include/vm
2137 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/sys \
2138     target=../../../../platform/sun4u/include/sys
2139 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/vm \
2140     target=../../../../platform/sun4u/include/vm
2141 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/sys \
2142     target=../../../../platform/sun4v/include/sys
2143 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/vm \
2144     target=../../../../platform/sun4v/include/vm
```

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

1

45632 Sat Aug 18 10:37:06 2012

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

dccp: manifest

```
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=bscbus alias=SUNW,bscbus
117 $(i386_ONLY)driver name=bscbus alias=SVI0101
118 $(sparc_ONLY)driver name=bscv alias=SUNW,bscv perms="* 0644 root sys"
119 $(i386_ONLY)driver name=bscv
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=dlistub 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=openeepr 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=physmem 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=vgtext \
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)/bscbus group=sys
337 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/bscv group=sys
338 file path=kernel/drv/$(ARCH64)/clone group=sys
339 file path=kernel/drv/$(ARCH64)/cn group=sys
340 file path=kernel/drv/$(ARCH64)/conskbd group=sys
341 file path=kernel/drv/$(ARCH64)/consms group=sys
342 file path=kernel/drv/$(ARCH64)/cpuid group=sys
343 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/cpunex group=sys
344 file path=kernel/drv/$(ARCH64)/crypto group=sys
345 file path=kernel/drv/$(ARCH64)/cryptoadm group=sys
346 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/dad group=sys
347 file path=kernel/drv/$(ARCH64)/dccp group=sys
348 file path=kernel/drv/$(ARCH64)/dccp6 group=sys
349 #endif /* ! codereview */
350 file path=kernel/drv/$(ARCH64)/devinfo group=sys
351 file path=kernel/drv/$(ARCH64)/dld group=sys
352 file path=kernel/drv/$(ARCH64)/dlpistub group=sys
353 file path=kernel/drv/$(ARCH64)/i8042 group=sys
354 file path=kernel/drv/$(ARCH64)/icmp group=sys
355 file path=kernel/drv/$(ARCH64)/icmp6 group=sys
356 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nb5000 group=sys
357 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nhm group=sys
358 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/intel_nhmex group=sys
359 file path=kernel/drv/$(ARCH64)/ip group=sys
360 file path=kernel/drv/$(ARCH64)/ip6 group=sys
361 file path=kernel/drv/$(ARCH64)/ipnet group=sys
362 file path=kernel/drv/$(ARCH64)/ippctl group=sys
363 file path=kernel/drv/$(ARCH64)/ipsecah group=sys
364 file path=kernel/drv/$(ARCH64)/ipsecesp group=sys
365 file path=kernel/drv/$(ARCH64)/iptun group=sys
366 file path=kernel/drv/$(ARCH64)/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/$(ARCH64)/llcl group=sys
372 file path=kernel/drv/$(ARCH64)/lofi group=sys
373 file path=kernel/drv/$(ARCH64)/log group=sys
374 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/mc-amd group=sys
375 file path=kernel/drv/$(ARCH64)/mm group=sys
376 file path=kernel/drv/$(ARCH64)/mouse8042 group=sys
377 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/mpt group=sys
378 file path=kernel/drv/$(ARCH64)/nulldriver group=sys
379 file path=kernel/drv/$(ARCH64)/openeepr group=sys
380 file path=kernel/drv/$(ARCH64)/options group=sys
381 file path=kernel/drv/$(ARCH64)/pci_pci group=sys
382 file path=kernel/drv/$(ARCH64)/pcieb group=sys
383 $(sparc_ONLY)file path=kernel/drv/$(ARCH64)/pcieb_bcm group=sys
384 file path=kernel/drv/$(ARCH64)/physmem group=sys
385 file path=kernel/drv/$(ARCH64)/poll group=sys
386 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/power group=sys
387 file path=kernel/drv/$(ARCH64)/pseudo group=sys
388 file path=kernel/drv/$(ARCH64)/ptc group=sys
389 file path=kernel/drv/$(ARCH64)/ptsl group=sys
390 file path=kernel/drv/$(ARCH64)/ramdisk group=sys
391 file path=kernel/drv/$(ARCH64)/random group=sys

```

```
392 file path=kernel/drv/$(ARCH64)/rts group=sys
393 file path=kernel/drv/$(ARCH64)/sad group=sys
394 file path=kernel/drv/$(ARCH64)/scsi_vhci group=sys
395 file path=kernel/drv/$(ARCH64)/sd group=sys
396 file path=kernel/drv/$(ARCH64)/sgen group=sys
397 file path=kernel/drv/$(ARCH64)/simnet group=sys
398 $(i386_ONLY)file path=kernel/drv/$(ARCH64)/smbios group=sys
399 file path=kernel/drv/$(ARCH64)/softmac group=sys
400 file path=kernel/drv/$(ARCH64)/spdsoc 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/openeepr group=sys
510 file path=kernel/drv/openeepr.conf group=sys
511 $(i386_ONLY)file path=kernel/drv/options group=sys
512 file path=kernel/drv/options.conf group=sys
513 $(i386_ONLY)file path=kernel/drv/pci_pci group=sys
514 $(i386_ONLY)file path=kernel/drv/pcieb group=sys
515 file path=kernel/drv/pcieb.conf group=sys
516 $(i386_ONLY)file path=kernel/drv/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/vgatest 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/$(ARCH64)/pcihip group=sys mode=0755
678 file path=kernel/misc/$(ARCH64)/pcmcia group=sys mode=0755
679 file path=kernel/misc/$(ARCH64)/rpcsec group=sys mode=0755
680 $(i386_ONLY)file path=kernel/misc/$(ARCH64)/sata group=sys mode=0755
681 file path=kernel/misc/$(ARCH64)/scsi group=sys mode=0755
682 file path=kernel/misc/$(ARCH64)/strplumb group=sys mode=0755
683 $(sparc_ONLY)file path=kernel/misc/$(ARCH64)/swapgeneric group=sys mode=0755
684 file path=kernel/misc/$(ARCH64)/tem group=sys mode=0755
685 file path=kernel/misc/$(ARCH64)/tlimod group=sys mode=0755
686 $(i386_ONLY)file path=kernel/misc/acpica group=sys mode=0755
687 $(i386_ONLY)file path=kernel/misc/agpmaster group=sys mode=0755
688 $(i386_ONLY)file path=kernel/misc/bignum group=sys mode=0755
689 $(i386_ONLY)file path=kernel/misc/bootdev group=sys mode=0755
690 $(i386_ONLY)file path=kernel/misc/busra group=sys mode=0755
691 $(i386_ONLY)file path=kernel/misc/cardbus group=sys mode=0755
692 $(i386_ONLY)file path=kernel/misc/cmlb group=sys mode=0755
693 $(i386_ONLY)file path=kernel/misc/consconfig group=sys mode=0755
694 $(i386_ONLY)file path=kernel/misc/ctf group=sys mode=0755
695 $(i386_ONLY)file path=kernel/misc/dls group=sys mode=0755
696 $(i386_ONLY)file path=kernel/misc/fssnap_if group=sys mode=0755
697 $(i386_ONLY)file path=kernel/misc/gld group=sys mode=0755
698 $(i386_ONLY)file path=kernel/misc/hook group=sys mode=0755
699 $(i386_ONLY)file path=kernel/misc/hpcsvc group=sys mode=0755
700 $(i386_ONLY)file path=kernel/misc/idmap group=sys mode=0755
701 $(i386_ONLY)file path=kernel/misc/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/pcihip 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/redimod 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)/ipsecesp \
869   target=../../kernel/drv/$(ARCH64)/ipsecesp
870 hardlink path=kernel/strmod/$(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/ipsecesp \
884   target=../../kernel/drv/ipsecesp
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/mdb/common/libstand/THIRDPARTYLICENSE \
901   license=usr/src/cmd/mdb/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
```

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

1

43092 Sat Aug 18 10:37:06 2012

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

dccp: starting module template

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 #
23 # Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright 2011 Nexenta Systems, Inc. All rights reserved.
25 # Copyright (c) 2012 by Delphix. All rights reserved.
26 #
27 #
28 #
29 # This Makefile defines all file modules for the directory uts/common
30 # and its children. These are the source files which may be considered
31 # common to all SunOS systems.
32 #
33 i386_CORE_OBJS += \
34     atomic.o      \
35     avintr.o      \
36     pic.o
37 #
38 sparc_CORE_OBJS +=
39 #
40 COMMON_CORE_OBJS += \
41     beep.o        \
42     bitset.o      \
43     bp_map.o      \
44     brand.o       \
45     cpucaps.o     \
46     cmt.o         \
47     cmt_policy.o  \
48     cpu.o         \
49     cpu_event.o   \
50     cpu_intr.o    \
51     cpu_pm.o      \
52     cpupart.o     \
53     cap_util.o    \
54     disp.o        \
55     group.o       \
56     kstat_fr.o    \
57     iscsiboot_prop.o \
58     lgrp.o        \
59     lgrp_topo.o   \
60     mmapobj.o     \
61     mutex.o
```

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

2

```
62     page_lock.o   \
63     page_retire.o \
64     panic.o       \
65     param.o       \
66     pg.o          \
67     pghw.o        \
68     putnext.o     \
69     rctl_proc.o   \
70     rwlock.o      \
71     seg_kmem.o    \
72     softint.o     \
73     string.o      \
74     strtol.o      \
75     strtoul.o     \
76     strtoll.o     \
77     strtoull.o    \
78     thread_intr.o \
79     vm_page.o     \
80     vm_pagelist.o \
81     zlib_obj.o    \
82     clock_tick.o
83 #
84 CORE_OBJS += $(COMMON_CORE_OBJS) $(MACH)_CORE_OBJS
85 #
86 ZLIB_OBJS = \
87     zutil.o zmod.o zmod_subr.o \
88     Adler32.o crc32.o deflate.o inflate.o \
89     inftrees.o trees.o
90 #
91 GENUNIX_OBJS += \
92     access.o \
93     acl.o \
94     acl_common.o \
95     adjtime.o \
96     alarm.o \
97     aio_subr.o \
98     auditsys.o \
99     audit_core.o \
100     audit_zone.o \
101     audit_memory.o \
102     autoconf.o \
103     avl.o \
104     bdev_dsort.o \
105     bio.o \
106     bitmap.o \
107     blabel.o \
108     brandsys.o \
109     bz2blocksort.o \
110     bz2compress.o \
111     bz2decompress.o \
112     bz2randtable.o \
113     bz2bzlib.o \
114     bz2crctable.o \
115     bz2huffman.o \
116     callb.o \
117     callout.o \
118     chdir.o \
119     chmod.o \
120     chown.o \
121     cladm.o \
122     class.o \
123     clock.o \
124     clock_highres.o \
125     clock_realtime.o \
126     close.o \
127     compress.o \
128     condvar.o
```

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

```

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

```

3

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

```

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

```

4

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

```

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

```

5

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

```

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

```

6

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

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

405 #
406 #     Stubs for the stand-alone linker/loader
407 #
408 sparc_GENSTUBS_OBJS = \
409     kobj_stubs.o

411 i386_GENSTUBS_OBJS =

413 COMMON_GENSTUBS_OBJS =

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

417 #
418 #     DTrace and DTrace Providers
419 #
420 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

422 SDT_OBJS += sdt_subr.o

424 PROFILE_OBJS += profile.o

426 SYSTRACE_OBJS += systrace.o

428 LOCKSTAT_OBJS += lockstat.o

430 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

432 DCPC_OBJS += dcpc.o

434 #
435 #     Driver (pseudo-driver) Modules
436 #
437 IPP_OBJS += ippctl.o

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

444 AUDIOEMU10K_OBJS += audioemu10k.o

446 AUDIOENS_OBJS += audioens.o

448 AUDIOVIA823X_OBJS += audiovia823x.o

450 AUDIOVIA97_OBJS += audiovia97.o

452 AUDIO1575_OBJS += audio1575.o

454 AUDIO810_OBJS += audio810.o

456 AUDIOCMI_OBJS += audiocmi.o
```

7

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

```
458 AUDIOCMIHD_OBJS += audiocmihd.o

460 AUDIOHD_OBJS += audiohd.o

462 AUDIOIXP_OBJS += audioixp.o

464 AUDIOLS_OBJS += audiols.o

466 AUDIOP16X_OBJS += audiop16x.o

468 AUDIOPCI_OBJS += audiopci.o

470 AUDIOSOLO_OBJS += audiosolo.o

472 AUDIOTS_OBJS += audits.o

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

476 BLKDEV_OBJS += blkdev.o

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

480 CONSKBD_OBJS += conskbd.o

482 CONSMS_OBJS += consms.o

484 OLDPPTY_OBJS += tty_ptyconf.o

486 PTC_OBJS += tty_pty.o

488 PTSL_OBJS += tty_pts.o

490 PTM_OBJS += ptm.o

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

495 PTS_OBJS += pts.o

497 PTY_OBJS += ptms_conf.o

499 SAD_OBJS += sad.o

501 MD4_OBJS += md4.o md4_mod.o

503 MD5_OBJS += md5.o md5_mod.o

505 SHA1_OBJS += sha1.o sha1_mod.o

507 SHA2_OBJS += sha2.o sha2_mod.o

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

512 DSCPMK_OBJS += dscpmk.o dscpmkddi.o

514 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

516 FLOWACCT_OBJS += flowacctddi.o flowacct.o

518 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

520 TSWTCL_OBJS += tswtcl.o tswtclddi.o

522 ARP_OBJS += arpddi.o
```

8


```

524 ICMP_OBJS +=      icmpddi.o
526 ICMP6_OBJS +=     icmp6ddi.o
528 RTS_OBJS +=       rtsddi.o

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

550 IP_OBJS +=          igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
551 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
552 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
553 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
554 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_queue.o \
555 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
556 ip_helper_stream.o ip_tunables.o \
557 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
558 conn_opt.o ip_attr.o ip_dce.o \
559 $(IP_ICMP_OBJS) \
560 $(IP_RTS_OBJS) \
561 $(IP_TCP_OBJS) \
562 $(IP_UDP_OBJS) \
563 $(IP_SCTP_OBJS) \
564 $(IP_ILB_OBJS) \
565 $(IP_DCCP_OBJS)
566 $(IP_ILB_OBJS)

567 IP6_OBJS +=         ip6ddi.o

569 HOOK_OBJS +=        hook.o

571 NETI_OBJS +=        neti_impl.o neti_mod.o neti_stack.o

573 KEYSOCK_OBJS +=     keysockddi.o keysock.o keysock_opt_data.o

575 IPNET_OBJS +=       ipnet.o ipnet_bpf.o

577 SPDSOCK_OBJS +=     spdsockddi.o spdsock.o spdsock_opt_data.o

579 IPSECESP_OBJS +=     ipsecespddi.o ipsecesp.o

581 IPSECAH_OBJS +=     ipsecahddi.o ipsecah.o sadb.o

583 SPPP_OBJS +=        sPPP.o sPPP_dlpi.o sPPP_mod.o s_common.o

585 SPPPTUN_OBJS +=     sppptun.o sppptun_mod.o

587 SPPPASYN_OBJS +=    spppasyn.o spppasyn_mod.o

```

```

589 SPPPCOMP_OBJS +=   spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
590 zlib.o

592 TCP_OBJS +=        tcpddi.o

594 TCP6_OBJS +=        tcp6ddi.o

596 NCA_OBJS +=         ncaddi.o

598 SDP SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

600 SCTP SOCK_MOD_OBJS += sockmod_sctp.o sockscdp.o sockscdpsubr.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 rdsrv3_ddi.o bind.o loop.o threads.o connection.o \
612 transport.o cong.o sysctl.o message.o rds_rcv.o send.o \
613 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
614 ib_rcv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
615 rdsrv3_sc.o rdsrv3_debug.o rdsrv3_impl.o rdma.o rdsrv3_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_srq.o ibtl_part.o
705 TAVOR_OBJS +=      tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
706                  tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
707                  tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
708                  tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o
710 HERMON_OBJS +=      hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
711                  hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
712                  hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
713                  hermon_srco.o hermon_stats.o hermon_umap.o hermon_wr.o \
714                  hermon_fcob.o hermon_fm.o
716 DAPLT_OBJS +=      daplt.o
718 SOL_OFS_OBJS +=      sol_cma.o sol_ib_cma.o sol_uobj.o \
719                  sol_ofs_debug_util.o sol_ofs_gen_util.o \
720                  sol_kverbs.o
```

```
722 SOL_UCMA_OBJS +=      sol_ucma.o
724 SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
725                  sol_uverbs_hca.o sol_uverbs_qp.o
727 SOL_UMAD_OBJS +=      sol_umad.o
729 KSTAT_OBJS +=      kstat.o
731 KSYMS_OBJS +=      ksyms.o
733 INSTANCE_OBJS +=      inst_sync.o
735 IWSCN_OBJS +=      iwscons.o
737 LOFI_OBJS +=      lofi.o LzmaDec.o
739 FSSNAP_OBJS +=      fssnap.o
741 FSSNAPIF_OBJS +=      fssnap_if.o
743 MM_OBJS +=      mem.o
745 PHYSMEM_OBJS +=      physmem.o
747 OPTIONS_OBJS +=      options.o
749 WINLOCK_OBJS +=      winlockio.o
751 PM_OBJS +=      pm.o
752 SRN_OBJS +=      srn.o
754 PSEUDO_OBJS +=      pseudonex.o
756 RAMDISK_OBJS +=      ramdisk.o
758 LLC1_OBJS +=      llc1.o
760 USBKBM_OBJS +=      usbkbm.o
762 USBWCM_OBJS +=      usbwcm.o
764 BOFI_OBJS +=      bofi.o
766 HID_OBJS +=      hid.o
768 HWA_RC_OBJS +=      hwarc.o
770 USBSKEL_OBJS +=      usbskel.o
772 USBVC_OBJS +=      usbvc.o usbvc_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_asynch.o \
847                s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
848                s1394_fa.o s1394_fcp.o \
849                s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o
851 HCI1394_OBJS +=      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 SYMSG_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 PCIHNP_OBJS += pcihnp.o

1033 OPENEPR_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 += fcsn.o
1055 FCTL_OBJS += fctl.o
1057 FP_OBJS += fp.o
1059 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
1060      ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o
1062 QLC_FW_2200_OBJS += ql_fw_2200.o
1064 QLC_FW_2300_OBJS += ql_fw_2300.o
1066 QLC_FW_2400_OBJS += ql_fw_2400.o
1068 QLC_FW_2500_OBJS += ql_fw_2500.o
1070 QLC_FW_6322_OBJS += ql_fw_6322.o
1072 QLC_FW_8100_OBJS += ql_fw_8100.o
1074 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o
1076 ZCONS_OBJS += zcons.o
1078 NV_SATA_OBJS += nv_sata.o
1080 SI3124_OBJS += si3124.o
1082 AHCI_OBJS += ahci.o
1084 PCIIDE_OBJS += pci-ide.o
1086 PCEPP_OBJS += pcepp.o
1088 CPC_OBJS += cpc.o
1090 CPUID_OBJS += cpuid_drv.o
1092 SYSEVENT_OBJS += sysevent.o
1094 BL_OBJS += bl.o
1096 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1097      drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1098      drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1099      drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1100      drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o
1102 FM_OBJS += devfm.o devfm_machdep.o
1104 RTLS_OBJS += rtls.o
1106 #
1107 #          exec modules
1108 #
1109 AOUTEXEC_OBJS += aout.o
1111 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o
1113 INTPEXEC_OBJS += intp.o
1115 SHBINEXEC_OBJS += shbin.o

```

```

1117 JAVAEXEC_OBJS += java.o
1119 #
1120 #          file system modules
1121 #
1122 AUTOFS_OBJS += auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o
1124 CACHEFS_OBJS += cachefs_cnode.o      cachefs_cod.o \
1125      cachefs_dir.o      cachefs_dlog.o cachefs_filegrp.o \
1126      cachefs_fscache.o  cachefs_ioctl.o cachefs_log.o \
1127      cachefs_module.o \
1128      cachefs_noopc.o      cachefs_resource.o \
1129      cachefs_strict.o \
1130      cachefs_subr.o      cachefs_vfsops.o \
1131      cachefs_vnops.o
1133 DCFS_OBJS += dc_vnops.o
1135 DEVFS_OBJS += devfs_subr.o      devfs_vfsops.o devfs_vnops.o
1137 DEV_OBJS += sdev_subr.o      sdev_vfsops.o sdev_vnops.o \
1138      sdev_ptsops.o sdev_zvolops.o sdev_comm.o \
1139      sdev_profile.o sdev_ncache.o sdev_netops.o \
1140      sdev_ipnetops.o \
1141      sdev_vtops.o
1143 CTFS_OBJS += ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1144      ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o
1146 OBJFS_OBJS += objfs_vfs.o      objfs_root.o objfs_common.o \
1147      objfs_odir.o      objfs_data.o
1149 FDFS_OBJS += fdops.o
1151 FIFO_OBJS += fifosubr.o      fifovnops.o
1153 PIPE_OBJS += pipe.o
1155 HSFS_OBJS += hsfs_node.o      hsfs_subr.o hsfs_vfsops.o hsfs_vnops.o \
1156      hsfs_susp.o      hsfs_rrip.o hsfs_susp_subr.o
1158 LOFS_OBJS += lofs_subr.o      lofs_vfsops.o lofs_vnops.o
1160 NAMEFS_OBJS += namevfs.o      namevno.o
1162 NFS_OBJS += nfs_client.o      nfs_common.o nfs_dump.o \
1163      nfs_subr.o      nfs_vfsops.o nfs_vnops.o \
1164      nfs_xdr.o      nfs_sys.o nfs_strerror.o \
1165      nfs3_vfsops.o  nfs3_vnops.o nfs3_xdr.o \
1166      nfs_acl_vnops.o nfs_acl_xdr.o nfs4_vfsops.o \
1167      nfs4_vnops.o  nfs4_xdr.o nfs4_idmap.o \
1168      nfs4_shadow.o  nfs4_subr.o \
1169      nfs4_attr.o    nfs4_rnode.o nfs4_client.o \
1170      nfs4_acache.o  nfs4_common.o nfs4_client_state.o \
1171      nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1172      nfs4_client_debug.o nfs_stats.o \
1173      nfs4_acl.o      nfs4_stub_vnops.o nfs_cmd.o
1175 NFSSRV_OBJS += nfs_server.o      nfs_srv.o nfs3_srv.o \
1176      nfs_acl_srv.o  nfs_auth.o nfs_auth_xdr.o \
1177      nfs_export.o   nfs_log.o  nfs_log_xdr.o \
1178      nfs4_srv.o     nfs4_state.o nfs4_srv_attr.o \
1179      nfs4_srv_ns.o  nfs4_db.o   nfs4_srv_deleg.o \
1180      nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o
1182 SMBSRV_SHARED_OBJS += \

```

```

1183         smb_inet.o \
1184         smb_match.o \
1185         smb_msgbuf.o \
1186         smb_oem.o \
1187         smb_string.o \
1188         smb_utf8.o \
1189         smb_door_legacy.o \
1190         smb_xdr.o \
1191         smb_token.o \
1192         smb_token_xdr.o \
1193         smb_sid.o \
1194         smb_native.o \
1195         smb_netbios_util.o

1197 SMBSRV_OBJS += $(SMBSRV_SHARED_OBJS) \
1198         smb_acl.o \
1199         smb_alloc.o \
1200         smb_close.o \
1201         smb_common_open.o \
1202         smb_common_transact.o \
1203         smb_create.o \
1204         smb_delete.o \
1205         smb_directory.o \
1206         smb_dispatch.o \
1207         smb_echo.o \
1208         smb_fem.o \
1209         smb_find.o \
1210         smb_flush.o \
1211         smb_fsinfo.o \
1212         smb_fsops.o \
1213         smb_init.o \
1214         smb_kdoor.o \
1215         smb_kshare.o \
1216         smb_kutil.o \
1217         smb_lock.o \
1218         smb_lock_byte_range.o \
1219         smb_locking_andx.o \
1220         smb_logoff_andx.o \
1221         smb_mangle_name.o \
1222         smb_mbuf_marshall.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 socktpi.o sockstr.o \
1279         sockcommon_vnops.o sockcommon_subr.o \
1280         sockcommon_sops.o sockcommon.o \
1281         sock_notsupp.o socknotify.o \
1282         nl7c.o nl7curi.o nl7chttp.o nl7clogd.o \
1283         nl7cnca.o sodirect.o sockfilter.o

1285 TMPFS_OBJS += tmp_dir.o tmp_subr.o tmp_tnode.o tmp_vfsops.o \
1286         tmp_vnops.o

1288 UDFS_OBJS += udf_alloc.o udf_bmap.o udf_dir.o \
1289         udf_inode.o udf_subr.o udf_vfsops.o \
1290         udf_vnops.o

1292 UFS_OBJS += ufs_alloc.o ufs_bmap.o ufs_dir.o ufs_xattr.o \
1293         ufs_inode.o ufs_subr.o ufs_tables.o ufs_vfsops.o \
1294         ufs_vnops.o quota.o quotacalls.o quota_ufs.o \
1295         ufs_filio.o ufs_lockfs.o ufs_thread.o ufs_trans.o \
1296         ufs_acl.o ufs_panic.o ufs_directio.o ufs_log.o \
1297         ufs_extvnops.o ufs_snap.o lufs.o lufs_thread.o \
1298         lufs_log.o lufs_map.o lufs_top.o lufs_debug.o \
1299         vscan_drv.o vscan_svc.o vscan_door.o

1301 NSMB_OBJS += smb_conn.o smb_dev.o smb_iod.o smb_pass.o \
1302         smb_rq.o smb_sign.o smb_smb.o smb_subrs.o \
1303         smb_time.o smb_tran.o smb_trantcp.o smb_usr.o \
1304         subr_mchain.o

1306 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1307 SMBFS_OBJS += smbfs_vfsops.o smbfs_vnops.o smbfs_node.o \
1308         smbfs_acl.o smbfs_client.o smbfs_smb.o \
1309         smbfs_subr.o smbfs_subr2.o \
1310         smbfs_rwlock.o smbfs_xattr.o \
1311         $(SMBFS_COMMON_OBJS)

1314 #

```

```

1315 #                LVM modules
1316 #
1317 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1318         md_med.o md_rename.o md_subr.o
1320 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o
1322 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o
1324 SOFTPART_OBJS += sp.o sp_ioctl.o
1326 STRIPE_OBJS += stripe.o stripe_ioctl.o
1328 HOTSPARES_OBJS += hotspares.o
1330 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o
1332 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o
1334 NOTIFY_OBJS += md_notify.o
1336 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o
1338 ZFS_COMMON_OBJS += \
1339     arc.o \
1340     bplist.o \
1341     bpobj.o \
1342     bptree.o \
1343     dbuf.o \
1344     ddt.o \
1345     ddt_zap.o \
1346     dmu.o \
1347     dmu_diff.o \
1348     dmu_send.o \
1349     dmu_object.o \
1350     dmu_objset.o \
1351     dmu_traverse.o \
1352     dmu_tx.o \
1353     dnode.o \
1354     dnode_sync.o \
1355     dsl_dir.o \
1356     dsl_dataset.o \
1357     dsl_deadlist.o \
1358     dsl_pool.o \
1359     dsl_synctask.o \
1360     dmu_zfetch.o \
1361     dsl_deleg.o \
1362     dsl_prop.o \
1363     dsl_scan.o \
1364     zfeature.o \
1365     gzip.o \
1366     lzjb.o \
1367     metaslab.o \
1368     refcount.o \
1369     sa.o \
1370     sha256.o \
1371     spa.o \
1372     spa_config.o \
1373     spa_errlog.o \
1374     spa_history.o \
1375     spa_misc.o \
1376     space_map.o \
1377     txg.o \
1378     uberblock.o \
1379     unique.o \
1380     vdev.o \

```

```

1381     vdev_cache.o \
1382     vdev_file.o \
1383     vdev_label.o \
1384     vdev_mirror.o \
1385     vdev_missing.o \
1386     vdev_queue.o \
1387     vdev_raidz.o \
1388     vdev_root.o \
1389     zap.o \
1390     zap_leaf.o \
1391     zap_micro.o \
1392     zfs_byteswap.o \
1393     zfs_debug.o \
1394     zfs_fm.o \
1395     zfs_fuid.o \
1396     zfs_sa.o \
1397     zfs_znode.o \
1398     zil.o \
1399     zio.o \
1400     zio_checksum.o \
1401     zio_compress.o \
1402     zio_inject.o \
1403     zle.o \
1404     zrlock.o
1406 ZFS_SHARED_OBJS += \
1407     zfeature_common.o \
1408     zfs_comutil.o \
1409     zfs_deleg.o \
1410     zfs_fletcher.o \
1411     zfs_namecheck.o \
1412     zfs_prop.o \
1413     zpool_prop.o \
1414     zprop_common.o
1416 ZFS_OBJS += \
1417     $(ZFS_COMMON_OBJS) \
1418     $(ZFS_SHARED_OBJS) \
1419     vdev_disk.o \
1420     zfs_acl.o \
1421     zfs_ctldir.o \
1422     zfs_dir.o \
1423     zfs_ioctl.o \
1424     zfs_log.o \
1425     zfs_onexit.o \
1426     zfs_replay.o \
1427     zfs_rlock.o \
1428     rrwlock.o \
1429     zfs_vfsops.o \
1430     zfs_vnops.o \
1431     zvol.o
1433 ZUT_OBJS += \
1434     zut.o
1436 #
1437 #                streams modules
1438 #
1439 BUFMOD_OBJS += bufmod.o
1441 CONNLD_OBJS += connld.o
1443 DEDUMP_OBJS += dedump.o
1445 DRCOMPAT_OBJS += drcompat.o

```

```

1447 LDLINUX_OBJS += ldlinux.o
1449 LDTERM_OBJS += ldterm.o uwidth.o
1451 PKKT_OBJS += pkt.o
1453 PFMOD_OBJS += pfmod.o
1455 PTEM_OBJS += ptem.o
1457 REDIRMOD_OBJS += strredirm.o
1459 TIMOD_OBJS += timod.o
1461 TIRDWR_OBJS += tirdwr.o
1463 TTCOMPAT_OBJS += ttcompat.o
1465 LOG_OBJS += log.o
1467 PIPEMOD_OBJS += pipemod.o
1469 RPCMOD_OBJS += rpcmod.o      clnt_cots.o      clnt_clts.o \
1470                  clnt_gen.o      clnt_perr.o      mt_rpcinit.o      rpc_calmsg.o \
1471                  rpc_prot.o      rpc_sztypes.o      rpc_subr.o      rpch_prot.o \
1472                  svc.o           svc_clts.o      svc_gen.o      svc_cots.o \
1473                  rpcsys.o      xdr_sizeof.o      clnt_rdma.o      svc_rdma.o \
1474                  xdr_rdma.o      rdma_subr.o      xdrdma_sizeof.o
1476 TLIMOD_OBJS += tlimod.o      t_kalloc.o      t_kbind.o      t_kclose.o \
1477                  t_kconnect.o      t_kfree.o      t_kgtstate.o      t_kopen.o \
1478                  t_krcvdat.o      t_ksndudat.o      t_kspoll.o      t_kunbind.o \
1479                  t_kutil.o
1481 RLMOD_OBJS += rlmmod.o
1483 TELMOD_OBJS += telmod.o
1485 CRYPTMOD_OBJS += cryptmod.o
1487 KB_OBJS += kbd.o      keytables.o
1489 #
1490 #           ID mapping module
1491 #
1492 IDMAP_OBJS += idmap_mod.o      idmap_kapi.o      idmap_xdr.o      idmap_cache.o
1494 #
1495 #           scheduling class modules
1496 #
1497 SDC_OBJS += sysdc.o
1499 RT_OBJS += rt.o
1500 RT_DPTBL_OBJS += rt_dptbl.o
1502 TS_OBJS += ts.o
1503 TS_DPTBL_OBJS += ts_dptbl.o
1505 IA_OBJS += ia.o
1507 FSS_OBJS += fss.o
1509 FX_OBJS += fx.o
1510 FX_DPTBL_OBJS += fx_dptbl.o
1512 #

```

```

1513 #           Inter-Process Communication (IPC) modules
1514 #
1515 IPC_OBJS += ipc.o
1517 IPCMSG_OBJS += msg.o
1519 IPCSEM_OBJS += sem.o
1521 IPCSHM_OBJS += shm.o
1523 #
1524 #           bignum module
1525 #
1526 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o
1528 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)
1530 #
1531 #           kernel cryptographic framework
1532 #
1533 KCF_OBJS += kcf.o kcf_callprov.o kcf_cbufoall.o kcf_cipher.o kcf_crypto.o \
1534             kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1535             kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1536             kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1537             kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1538             kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1539             fips_random.o
1541 CRYPTOADM_OBJS += cryptoadm.o
1543 CRYPTO_OBJS += crypto.o
1545 DPROV_OBJS += dprov.o
1547 DCA_OBJS += dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1548             dca_rsa.o
1550 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o
1552 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o
1554 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o
1556 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1557             ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1558             ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1559             ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1560             mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1561             secitem.o ec2_test.o ecp_test.o
1563 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o
1565 SWRANDPROV_OBJS += swrand.o
1567 #
1568 #           kernel SSL
1569 #
1570 KSSL_OBJS += kssl.o ksslioc1.o
1572 KSSL_SOCKETFIL_MOD_OBJS += ksslfilter.o ksslap1.o ksslrec.o
1574 #
1575 #           misc. modules
1576 #
1578 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \

```



```

1579      audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1580      audit_mem.o

1582 PCIC_OBJS +=      pcic.o

1584 RPCSEC_OBJS +=      secmod.o          sec_clnt.o          sec_svc.o          sec_gen.o \
1585      auth_des.o      auth_kern.o      auth_none.o      auth_loopb.o \
1586      authdesprt.o    authdesubr.o    authu_prot.o \
1587      key_call.o      key_prot.o      svc_authu.o      svcauthdes.o

1589 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1590      rpcsec_gss_utils.o svc_rpcsec_gss.o

1592 CONSCONFIG_OBJS +=      consconfig.o

1594 CONSCONFIG_DACF_OBJS +=      consconfig_dacf.o consplat.o

1596 TEM_OBJS +=      tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1598 KBTRANS_OBJS +=      \
1599      kbtrans.o          \
1600      kbtrans_keytables.o \
1601      kbtrans_polled.o   \
1602      kbtrans_streams.o  \
1603      usb_keytables.o

1605 KGSSD_OBJS +=      gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1606      gss_display_name.o gss_release_name.o gss_import_name.o \
1607      gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1609 KGSSD_DERIVED_OBJS = gssd_xdr.o

1611 KGSS_DUMMY_OBJS +=      dmech.o

1613 KSOCKET_OBJS +=      ksocket.o ksocket_mod.o

1615 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1616      nfold.o verify_checksum.o prng.o block_size.o make_checksum.o \
1617      checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1619 # crypto/des
1620 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1622 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1624 CRYPTO_ARCFOUR= k5_arcfour.o

1626 # crypto/enc_provider
1627 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1629 # crypto/hash_provider
1630 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_kshal.o

1632 # crypto/keyhash_provider
1633 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1635 # crypto/crc32
1636 CRYPTO_CRC32= crc32.o

1638 # crypto/old
1639 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1641 # crypto/raw
1642 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1644 K5_KRB= kfree.o copy_key.o \

```

```

1645      parse.o init_ctx.o \
1646      ser_adata.o ser_addr.o \
1647      ser_auth.o ser_cksum.o \
1648      ser_key.o ser_princ.o \
1649      serialize.o unparse.o \
1650      ser_actx.o

1652 K5_OS=      timeofday.o toffset.o \
1653      init_os_ctx.o c_ustime.o

1655 SEAL=
1656 # EXPORT DELETE START
1657 SEAL=      seal.o unseal.o
1658 # EXPORT DELETE END

1660 MECH=      delete_sec_context.o \
1661      import_sec_context.o \
1662      gssapi_krb5.o \
1663      k5seal.o k5unseal.o k5sealv3.o \
1664      ser_sctx.o \
1665      sign.o \
1666      util_crypt.o \
1667      util_validate.o util_ordering.o \
1668      util_seqnum.o util_set.o util_seed.o \
1669      wrap_size_limit.o verify.o

1673 MECH_GEN=      util_token.o

1676 KGSS_KRB5_OBJS +=      krb5mech.o \
1677      $(MECH) $(SEAL) $(MECH_GEN) \
1678      $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1679      $(CRYPTO_ENC) $(CRYPTO_HASH) \
1680      $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1681      $(CRYPTO_OLD) \
1682      $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1684 DES_OBJS +=      des_crypt.o des_impl.o des_ks.o des_soft.o

1686 DLBOOT_OBJS +=      bootparam_xdr.o nfs_dlinet.o scan.o

1688 KRTLD_OBJS +=      kobj_bootflags.o getoptstr.o \
1689      kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1691 MOD_OBJS +=      modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1693 STRPLUMB_OBJS +=      strplumb.o

1695 CPR_OBJS +=      cpr_driver.o cpr_dump.o \
1696      cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1697      cpr_uthread.o

1699 PROF_OBJS +=      prf.o

1701 SE_OBJS +=      se_driver.o

1703 SYSACCT_OBJS +=      acct.o

1705 ACCTCTL_OBJS +=      acctctl.o

1707 EXACCTSYS_OBJS +=      exacctsyes.o

1709 KAIO_OBJS +=      aio.o

```

```
1711 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o
1713 BUSRA_OBJS += busra.o
1715 PCS_OBJS += pcs.o
1717 PCAN_OBJS += pcan.o
1719 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o
1721 PCSER_OBJS += pcser.o pcser_cis.o
1723 PCWL_OBJS += pcwl.o
1725 PSET_OBJS += pset.o
1727 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o
1729 UHCI_OBJS += uhci.o uhciutil.o uhci_tgt.o uhcihub.o uhci_polled.o
1731 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o
1733 HUBD_OBJS += hubd.o
1735 USB_MID_OBJS += usb_mid.o
1737 USB_IA_OBJS += usb_ia.o
1739 UWBA_OBJS += uwba.o uwba_i.o
1741 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o
1743 HWAHC_OBJS += hwahc.o hwahc_util.o
1745 WUSB_DF_OBJS += wusb_df.o
1746 WUSB_FWMOD_OBJS += wusb_fwmod.o
1748 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1749             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1750             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o
1752 IBD_OBJS += ibd.o ibd_cm.o
1754 EIBNX_OBJS += enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1755             enx_misc.o enx_q.o enx_ctl.o
1757 EOIB_OBJS += eib_adm.o eib_chan.o eib_cmnm.o eib_ctl.o eib_data.o \
1758             eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1759             eib_rsrc.o eib_svc.o eib_vnic.o
1761 DLPSTUB_OBJS += dlpstub.o
1763 SDP_OBJS += sdpddi.o
1765 TRILL_OBJS += trill.o
1767 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1768             ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o
1770 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o
1772 RPCIB_OBJS += rpcib.o
1774 KMDB_OBJS += kdrv.o
1776 AFE_OBJS += afe.o
```

```
1778 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1779             bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o
1781 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o
1783 EFE_OBJS += efe.o
1785 ELXL_OBJS += elxl.o
1787 HME_OBJS += hme.o
1789 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1790             ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o
1792 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1793             nge_log.o nge_rx.o nge_tx.o nge_xmii.o
1795 PCN_OBJS += pcn.o
1797 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o
1799 URTW_OBJS += urtw.o
1801 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_anis.o arn_phy.o arn_
1802             arn_main.o arn_recv.o arn_xmit.o arn_rc.o
1804 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o
1806 ATU_OBJS += atu.o
1808 IPW_OBJS += ipw2100_hw.o ipw2100.o
1810 IWI_OBJS += ipw2200_hw.o ipw2200.o
1812 IWH_OBJS += iwh.o
1814 IWK_OBJS += iwk2.o
1816 IWP_OBJS += iwp.o
1818 MWL_OBJS += mwl.o
1820 MWLFW_OBJS += mwlfw_mode.o
1822 WPI_OBJS += wpi.o
1824 RAL_OBJS += rt2560.o ral_rate.o
1826 RUM_OBJS += rum.o
1828 RWD_OBJS += rt2661.o
1830 RWN_OBJS += rt2860.o
1832 UATH_OBJS += uath.o
1834 UATHFW_OBJS += uathfw_mod.o
1836 URAL_OBJS += ural.o
1838 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o
1840 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o
1842 MXFE_OBJS += mxfe.o
```

```
1844 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o
1846 SFE_OBJS += sfe.o sfe_util.o
1848 BFE_OBJS += bfe.o
1850 BRIDGE_OBJS += bridge.o
1852 IDM_SHARED_OBJS += base64.o
1854 IDM_OBJS += $(IDM_SHARED_OBJS) \
1855             idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o
1857 VR_OBJS += vr.o
1859 ATGE_OBJS += atge_main.o atge_lle.o atge_mii.o atge_ll.o
1861 YGE_OBJS = yge.o
1863 #
1864 #       Build up defines and paths.
1865 #
1866 LINT_DEFS      += -Dunix
1868 #
1869 #       This duality can be removed when the native and target compilers
1870 #       are the same (or at least recognize the same command line syntax!)
1871 #       It is a bug in the current compilation system that the assembler
1872 #       can't process the -Y I, flag.
1873 #
1874 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1875 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1876 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1878 PCIEB_OBJS += pcieb.o
1880 #       Chelsio N110 10G NIC driver module
1881 #
1882 CH_OBJS = ch.o glue.o pe.o sge.o
1884 CH_COM_OBJS = ch_mac.o ch_subr.o cspi.o espi.o ixfl010.o mc3.o mc4.o mc5.o \
1885             mv88elxxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1886             vsc7321.o vsc7326.o xpak.o
1888 #
1889 #       PCI strings file
1890 #
1891 PCI_STRING_OBJS = pci_strings.o
1893 NET_DACF_OBJS += net_dacf.o
1895 #
1896 #       Xframe 10G NIC driver module
1897 #
1898 XGE_OBJS = xge.o xgell.o
1900 XGE_HAL_OBJS = xgehal-channel.o xgehal-fifo.o xgehal-ring.o xgehal-config.o \
1901             xgehal-driver.o xgehal-mm.o xgehal-stats.o xgehal-device.o \
1902             xge-queue.o xgehal-mgmt.o xgehal-mgmtaux.o
1904 #
1905 #       e1000g module
1906 #
1907 E1000G_OBJS += e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1908             e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
```

```
1909             e1000_mac.o e1000_manage.o e1000_nvmm.o e1000_osdep.o \
1910             e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1911             e1000g_tx.o e1000g_rx.o e1000g_stat.o
1913 #
1914 #       Intel 82575 1G NIC driver module
1915 #
1916 IGB_OBJS = igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1917             igb_nvmm.o igb_osdep.o igb_phy.o igb_buf.o \
1918             igb_debug.o igb_gld.o igb_log.o igb_main.o \
1919             igb_rx.o igb_stat.o igb_tx.o
1921 #
1922 #       Intel Pro/100 NIC driver module
1923 #
1924 IPRB_OBJS = iprb.o
1926 #
1927 #       Intel 10GbE PCIE NIC driver module
1928 #
1929 IXGBE_OBJS = ixgbe_82598.o ixgbe_82599.o ixgbe_api.o \
1930             ixgbe_common.o ixgbe_phy.o \
1931             ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o \
1932             ixgbe_log.o ixgbe_main.o \
1933             ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o \
1934             ixgbe_tx.o
1936 #
1937 #       NIU 10G/1G driver module
1938 #
1939 NXGE_OBJS = nxge_mac.o nxge_ipp.o nxge_rxdma.o \
1940             nxge_txdma.o nxge_txc.o nxge_main.o \
1941             nxge_hw.o nxge_fzc.o nxge_virtual.o \
1942             nxge_send.o nxge_classify.o nxge_fflp.o \
1943             nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o \
1944             nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o \
1945             nxge_hio.o nxge_hio_guest.o nxge_intr.o
1947 NXGE_NPI_OBJS = \
1948             npi.o npi_mac.o npi_ipp.o \
1949             npi_txdma.o npi_rxdma.o npi_txc.o \
1950             npi_zcp.o npi_espc.o npi_fflp.o \
1951             npi_vir.o
1953 NXGE_HCALL_OBJS = \
1954             nxge_hcall.o
1956 #
1957 #       kiconv modules
1958 #
1959 KICONV_EMEA_OBJS += kiconv_emea.o
1961 KICONV_JA_OBJS += kiconv_ja.o
1963 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o
1965 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o
1967 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o
1969 #
1970 #       AAC module
1971 #
1972 AAC_OBJS = aac.o aac_ioctl.o
1974 #
```

```
1975 #         sdcard modules
1976 #
1977 SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
1978 SDHOST_OBJS =   sdhost.o

1980 #
1981 #         hxge 10G driver module
1982 #
1983 HXGE_OBJS =      hxge_main.o hxge_vmac.o hxge_send.o          \
1984                  hxge_txdma.o hxge_rxdma.o hxge_virtual.o     \
1985                  hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o \
1986                  hxge_ndd.o hxge_pfc.o                       \
1987                  hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o     \
1988                  hpi_vir.o hpi_pfc.o

1990 #
1991 #         MEGARAID_SAS module
1992 #
1993 MEGA_SAS_OBJS =  megaraid_sas.o

1995 #
1996 #         MR_SAS module
1997 #
1998 MR_SAS_OBJS =    mr_sas.o

2000 #
2001 #         ISCSI_INITIATOR module
2002 #
2003 ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o      \
2004                          iscsi_ioctl.o iscsid.o iscsi.o        \
2005                          iscsi_login.o isns_client.o iscsiAuthClient.o \
2006                          iscsi_lun.o iscsiAuthClientGlue.o     \
2007                          iscsi_net.o nvfile.o iscsi_cmd.o      \
2008                          iscsi_queue.o persistent.o iscsi_conn.o \
2009                          iscsi_sess.o radius_auth.o iscsi_crc.o \
2010                          iscsi_stats.o radius_packet.o iscsi_doorclt.o \
2011                          iscsi_targetparam.o utils.o kifconf.o

2013 #
2014 #         ntxn 10Gb/1Gb NIC driver module
2015 #
2016 NTXN_OBJS =      unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
2017                  unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

2019 #
2020 #         Myricom 10Gb NIC driver module
2021 #
2022 MYRI10GE_OBJS =  myri10ge.o myri10ge_lro.o

2024 #
2025 #         nulldriver module
2026 NULLDRIVER_OBJS =      nulldriver.o

2028 TPM_OBJS =            tpm.o tpm_hcall.o
```

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

1

72399 Sat Aug 18 10:37:07 2012

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

dccp: starting module template

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
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 $@ $<
```

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

2

```
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)/exacct/%.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/cacheefs/%.c
213 $(COMPILE.c) -o $@ $<
214 $(CTFCONVERT_O)

216 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/dcf/%.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/fifo/%.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 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/nfs/%.c
261 $(COMPILE.c) -o $@ $<
262 $(CTFCONVERT_O)

264 $(OBJSDIR)/%.o: $(COMMONBASE)/smbsrv/%.c
265 $(COMPILE.c) -o $@ $<
266 $(CTFCONVERT_O)

268 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/smbsrv/%.c
269 $(COMPILE.c) -o $@ $<
270 $(CTFCONVERT_O)

272 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/objfs/%.c
273 $(COMPILE.c) -o $@ $<
274 $(CTFCONVERT_O)

276 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/pcfs/%.c
277 $(COMPILE.c) -o $@ $<
278 $(CTFCONVERT_O)

280 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/portfs/%.c
281 $(COMPILE.c) -o $@ $<
282 $(CTFCONVERT_O)

284 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/proc/%.c
285 $(COMPILE.c) -o $@ $<
286 $(CTFCONVERT_O)

288 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/sharefs/%.c
289 $(COMPILE.c) -o $@ $<
290 $(CTFCONVERT_O)

292 $(OBJSDIR)/%.o: $(COMMONBASE)/smbclnt/%.c
293 $(COMPILE.c) -o $@ $<
294 $(CTFCONVERT_O)

296 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/smbclnt/net smb/%.c
297 $(COMPILE.c) -o $@ $<
298 $(CTFCONVERT_O)

300 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/smbclnt/smbfs/%.c
301 $(COMPILE.c) -o $@ $<
302 $(CTFCONVERT_O)

304 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/sockfs/%.c
305 $(COMPILE.c) -o $@ $<
306 $(CTFCONVERT_O)

308 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/specfs/%.c
309 $(COMPILE.c) -o $@ $<
310 $(CTFCONVERT_O)

312 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/swapfs/%.c
313 $(COMPILE.c) -o $@ $<
314 $(CTFCONVERT_O)

316 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/tmpfs/%.c
317 $(COMPILE.c) -o $@ $<
318 $(CTFCONVERT_O)

320 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/udfs/%.c
321 $(COMPILE.c) -o $@ $<
322 $(CTFCONVERT_O)

324 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/uifs/%.c
325 $(COMPILE.c) -o $@ $<

```

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

6

```

326 $(CTFCONVERT_O)

328 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/vscan/%.c
329 $(COMPILE.c) -o $@ $<
330 $(CTFCONVERT_O)

332 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/zfs/%.c
333 $(COMPILE.c) -o $@ $<
334 $(CTFCONVERT_O)

336 $(OBJSDIR)/%.o: $(UTSBASE)/common/fs/zut/%.c
337 $(COMPILE.c) -o $@ $<
338 $(CTFCONVERT_O)

340 $(OBJSDIR)/%.o: $(COMMONBASE)/xattr/%.c
341 $(COMPILE.c) -o $@ $<
342 $(CTFCONVERT_O)

344 $(OBJSDIR)/%.o: $(COMMONBASE)/zfs/%.c
345 $(COMPILE.c) -o $@ $<
346 $(CTFCONVERT_O)

348 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/pmcs/%.c
349 $(COMPILE.c) -o $@ $<
350 $(CTFCONVERT_O)

352 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/scsi/adapters/pmcs/%.bin
353 $(COMPILE.b) -o $@ $<
354 $(CTFCONVERT_O)

356 $(OBJSDIR)/%.o: $(COMMONBASE)/fsreparse/%.c
357 $(COMPILE.c) -o $@ $<
358 $(CTFCONVERT_O)

360 KMECHKRB5_BASE=$(UTSBASE)/common/gssapi/mechs/krb5

362 KGSSDFFLAGS=-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 KGSSDFFLAGS += $(KRB5_DEFS)

369 $(OBJSDIR)/%.o: $(UTSBASE)/common/gssapi/%.c
370 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
371 $(CTFCONVERT_O)

373 $(OBJSDIR)/%.o: $(UTSBASE)/common/gssapi/mechs/dummy/%.c
374 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
375 $(CTFCONVERT_O)

377 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/%.c
378 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
379 $(CTFCONVERT_O)

381 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/%.c
382 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
383 $(CTFCONVERT_O)

385 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/des/%.c
386 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
387 $(CTFCONVERT_O)

389 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/crypto/arcfour/%.c
390 $(COMPILE.c) $(KGSSDFFLAGS) -o $@ $<
391 $(CTFCONVERT_O)

```

```

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

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

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

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

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

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

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

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

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

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

431 $(OBJSDIR)/%.o: $(KMECHKRB5_BASE)/profile/%.c
432 $(COMPILE.c) $(KGSSDFFLAGS) -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/scs1394/%.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/rds3/%.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/ocel/%.c
1100 $(COMPILE.c) -o $@ $<
1101 $(CTFCONVERT_O)

1103 $(OBJSDIR)/%.o: $(UTSBASE)/common/io/fibre-channel/fca/fcoe/%.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/el000g/%.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/myril0ge/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

```

```

1448          $(COMPILE.c) -o $@ $<
1449          $(CTFCONVERT_O)

1451 $(OBJSDIR)/%.o:          $(COMMONBASE)/net/dhcp%.c
1452          $(COMPILE.c) -o $@ $<
1453          $(CTFCONVERT_O)

1455 $(OBJSDIR)/%.o:          $(COMMONBASE)/nvpair%.c
1456          $(COMPILE.c) -o $@ $<
1457          $(CTFCONVERT_O)

1459 $(OBJSDIR)/%.o:          $(UTSBASE)/common/os%.c
1460          $(COMPILE.c) -o $@ $<
1461          $(CTFCONVERT_O)

1463 $(OBJSDIR)/%.o:          $(UTSBASE)/common/pcmcia/cis%.c
1464          $(COMPILE.c) -o $@ $<
1465          $(CTFCONVERT_O)

1467 $(OBJSDIR)/%.o:          $(UTSBASE)/common/pcmcia/cs%.c
1468          $(COMPILE.c) -o $@ $<
1469          $(CTFCONVERT_O)

1471 $(OBJSDIR)/%.o:          $(UTSBASE)/common/pcmcia/nexus%.c
1472          $(COMPILE.c) -o $@ $<
1473          $(CTFCONVERT_O)

1475 $(OBJSDIR)/%.o:          $(UTSBASE)/common/pcmcia/pcs%.c
1476          $(COMPILE.c) -o $@ $<
1477          $(CTFCONVERT_O)

1479 $(OBJSDIR)/%.o:          $(UTSBASE)/common/rpc%.c
1480          $(COMPILE.c) -o $@ $<
1481          $(CTFCONVERT_O)

1483 $(OBJSDIR)/%.o:          $(UTSBASE)/common/rpc/sec%.c
1484          $(COMPILE.c) -o $@ $<
1485          $(CTFCONVERT_O)

1487 $(OBJSDIR)/%.o:          $(UTSBASE)/common/rpc/sec_gss%.c
1488          $(COMPILE.c) -o $@ $<
1489          $(CTFCONVERT_O)

1491 $(OBJSDIR)/%.o:          $(COMMONBASE)/crypto/sha1%.c
1492          $(COMPILE.c) -o $@ $<
1493          $(CTFCONVERT_O)

1495 $(OBJSDIR)/%.o:          $(COMMONBASE)/crypto/sha2%.c
1496          $(COMPILE.c) -o $@ $<
1497          $(CTFCONVERT_O)

1499 $(OBJSDIR)/%.o:          $(UTSBASE)/common/syscall%.c
1500          $(COMPILE.c) -o $@ $<
1501          $(CTFCONVERT_O)

1503 $(OBJSDIR)/%.o:          $(UTSBASE)/common/tnf%.c
1504          $(COMPILE.c) -o $@ $<
1505          $(CTFCONVERT_O)

1507 $(OBJSDIR)/%.o:          $(COMMONBASE)/tsol%.c
1508          $(COMPILE.c) -o $@ $<
1509          $(CTFCONVERT_O)

1511 $(OBJSDIR)/%.o:          $(COMMONBASE)/util%.c
1512          $(COMPILE.c) -o $@ $<
1513          $(CTFCONVERT_O)

```

[illegible]


```

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) $(KGSSD_FLAGS) $< $(LTAIL))
1828 $(LINTS_DIR)/%.ln:      $(UTSBASE)/common/gssapi/mechs/dummy/%.c
1829   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1831 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/%.c
1832   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1834 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/%.c
1835   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1837 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/des/%.c
1838   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1840 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/dk/%.c
1841   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1843 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/os/%.c

```

```

1844      @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1846 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/arcfour/%.c
1847   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1849 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/enc_provider/%.c
1850   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1852 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/hash_provider/%.c
1853   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1855 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/keyhash_provider/%.c
1856   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1858 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/raw/%.c
1859   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1861 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/crypto/old/%.c
1862   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1864 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/krb5/krb/%.c
1865   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1867 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/krb5/os/%.c
1868   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(LTAIL))
1870 $(LINTS_DIR)/%.ln:      $(KMECHKRB5_BASE)/mech/%.c
1871   @$(LHEAD) $(LINT.c) $(KGSSD_FLAGS) $< $(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) $(IPF_FLAGS) $< $(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) $(IPF_FLAGS) $< $(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: $(UTSBASE)/common/io/audio/drv/audiopl6x/%.c
2001     @$(LHEAD) $(LINT.c) $< $(LTAIL))

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

2006 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/audio/drv/audiotst/%.c
2007     @$(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: $(UTSBASE)/common/io/elxl/%.c
2073     @$(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: $(UTSBASE)/common/io/lvm/strip/%.c
2202     @$(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: $(UTSBASE)/common/io/ppp/spppasyn/%.c
2271     @$(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: $(UTSBASE)/common/io/usb/clients/audio/usb_ac/%.c
2388     @$(LHEAD) $(LINT.c) $< $(LTAIL))

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

2393 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/audio/usb_ah/%.c
2394     @$(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: $(UTSBASE)/common/io/usb/clients/usbbkm/%.c
2412     @$(LHEAD) $(LINT.c) $< $(LTAIL))

2414 $(LINTSDIR)/%.ln: $(UTSBASE)/common/io/usb/clients/usbsms/%.c
2415     @$(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: $(COMMONBASE)/crypto/sha1/%.c
2622     @$(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))
```

57591 Sat Aug 18 10:37:07 2012

new/usr/src/uts/common/dtrace/sdt_subr.c

dccp: basic dtrace

_____unchanged_portion_omitted_____

```

99 sdt_provider_t sdt_providers[] = {
100     "vtrace", "__vtrace_", &vtrace_attr, 0 },
101     "sysinfo", "__cpu_sysinfo_", &info_attr, 0 },
102     "vminfo", "__cpu_vminfo_", &info_attr, 0 },
103     "fpuinfo", "__fpuinfo_", &fpu_attr, 0 },
104     "sched", "__sched_", &stab_attr, 0 },
105     "proc", "__proc_", &stab_attr, 0 },
106     "io", "__io_", &stab_attr, 0 },
107     "ip", "__ip_", &stab_attr, 0 },
108     "tcp", "__tcp_", &stab_attr, 0 },
109     "udp", "__udp_", &stab_attr, 0 },
110     "dccp", "__dccp_", &stab_attr, 0 },
111 #endif /* ! codereview */
112     "mib", "__mib_", &stab_attr, 0 },
113     "fsinfo", "__fsinfo_", &fsinfo_attr, 0 },
114     "iscsi", "__iscsi_", &iscsi_attr, 0 },
115     "nfsv3", "__nfsv3_", &stab_attr, 0 },
116     "nfsv4", "__nfsv4_", &stab_attr, 0 },
117     "xpv", "__xpv_", &xpv_attr, 0 },
118     "fc", "__fc_", &fc_attr, 0 },
119     "srp", "__srp_", &fc_attr, 0 },
120     "sysevent", "__sysevent_", &stab_attr, 0 },
121     "sdt", NULL, &sdt_attr, 0 },
122     NULL }
123 };

125 sdt_argdesc_t sdt_args[] = {
126     "sched", "wakeup", 0, 0, "kthread_t **", "lwpsinfo_t **",
127     "sched", "wakeup", 1, 0, "kthread_t **", "psinfo_t **",
128     "sched", "dequeue", 0, 0, "kthread_t **", "lwpsinfo_t **",
129     "sched", "dequeue", 1, 0, "kthread_t **", "psinfo_t **",
130     "sched", "dequeue", 2, 1, "disp_t **", "cpuinfo_t **",
131     "sched", "enqueue", 0, 0, "kthread_t **", "lwpsinfo_t **",
132     "sched", "enqueue", 1, 0, "kthread_t **", "psinfo_t **",
133     "sched", "enqueue", 2, 1, "disp_t **", "cpuinfo_t **",
134     "sched", "enqueue", 3, 2, "int" },
135     "sched", "off-cpu", 0, 0, "kthread_t **", "lwpsinfo_t **",
136     "sched", "off-cpu", 1, 0, "kthread_t **", "psinfo_t **",
137     "sched", "tick", 0, 0, "kthread_t **", "lwpsinfo_t **",
138     "sched", "tick", 1, 0, "kthread_t **", "psinfo_t **",
139     "sched", "change-pri", 0, 0, "kthread_t **", "lwpsinfo_t **",
140     "sched", "change-pri", 1, 0, "kthread_t **", "psinfo_t **",
141     "sched", "change-pri", 2, 1, "pri_t" },
142     "sched", "schedctl-nopreempt", 0, 0, "kthread_t **", "lwpsinfo_t **",
143     "sched", "schedctl-nopreempt", 1, 0, "kthread_t **", "psinfo_t **",
144     "sched", "schedctl-nopreempt", 2, 1, "int" },
145     "sched", "schedctl-preempt", 0, 0, "kthread_t **", "lwpsinfo_t **",
146     "sched", "schedctl-preempt", 1, 0, "kthread_t **", "psinfo_t **",
147     "sched", "schedctl-yield", 0, 0, "int" },
148     "sched", "surrender", 0, 0, "kthread_t **", "lwpsinfo_t **",
149     "sched", "surrender", 1, 0, "kthread_t **", "psinfo_t **",
150     "sched", "cpucaps-sleep", 0, 0, "kthread_t **", "lwpsinfo_t **",
151     "sched", "cpucaps-sleep", 1, 0, "kthread_t **", "psinfo_t **",
152     "sched", "cpucaps-wakeup", 0, 0, "kthread_t **", "lwpsinfo_t **",
153     "sched", "cpucaps-wakeup", 1, 0, "kthread_t **", "psinfo_t **",

155     "proc", "create", 0, 0, "proc_t **", "psinfo_t **",
156     "proc", "exec", 0, 0, "string" },
157     "proc", "exec-failure", 0, 0, "int" },

```

```

158     "proc", "exit", 0, 0, "int" },
159     "proc", "fault", 0, 0, "int" },
160     "proc", "fault", 1, 1, "siginfo_t **",
161     "proc", "lwp-create", 0, 0, "kthread_t **", "lwpsinfo_t **",
162     "proc", "lwp-create", 1, 0, "kthread_t **", "psinfo_t **",
163     "proc", "signal-clear", 0, 0, "int" },
164     "proc", "signal-clear", 1, 1, "siginfo_t **",
165     "proc", "signal-discard", 0, 0, "kthread_t **", "lwpsinfo_t **",
166     "proc", "signal-discard", 1, 1, "proc_t **", "psinfo_t **",
167     "proc", "signal-discard", 2, 2, "int" },
168     "proc", "signal-handle", 0, 0, "int" },
169     "proc", "signal-handle", 1, 1, "siginfo_t **",
170     "proc", "signal-handle", 2, 2, "void (*) (void)" },
171     "proc", "signal-send", 0, 0, "kthread_t **", "lwpsinfo_t **",
172     "proc", "signal-send", 1, 0, "kthread_t **", "psinfo_t **",
173     "proc", "signal-send", 2, 1, "int" },

175     "io", "start", 0, 0, "buf_t **", "bufinfo_t **",
176     "io", "start", 1, 0, "buf_t **", "devinfo_t **",
177     "io", "start", 2, 0, "buf_t **", "fileinfo_t **",
178     "io", "done", 0, 0, "buf_t **", "bufinfo_t **",
179     "io", "done", 1, 0, "buf_t **", "devinfo_t **",
180     "io", "done", 2, 0, "buf_t **", "fileinfo_t **",
181     "io", "wait-start", 0, 0, "buf_t **", "bufinfo_t **",
182     "io", "wait-start", 1, 0, "buf_t **", "devinfo_t **",
183     "io", "wait-start", 2, 0, "buf_t **", "fileinfo_t **",
184     "io", "wait-done", 0, 0, "buf_t **", "bufinfo_t **",
185     "io", "wait-done", 1, 0, "buf_t **", "devinfo_t **",
186     "io", "wait-done", 2, 0, "buf_t **", "fileinfo_t **",

188     { "mib", NULL, 0, 0, "int" },

190     { "fsinfo", NULL, 0, 0, "vnode_t **", "fileinfo_t **",
191     { "fsinfo", NULL, 1, 1, "int", "int" },

193     { "iscsi", "async-send", 0, 0, "idm_conn_t **", "conninfo_t **",
194     { "iscsi", "async-send", 1, 1, "iscsi_async_evt_hdr_t **",
195     { "iscsiinfo_t **",
196     { "iscsi", "login-command", 0, 0, "idm_conn_t **", "conninfo_t **",
197     { "iscsi", "login-command", 1, 1, "iscsi_login_hdr_t **",
198     { "iscsiinfo_t **",
199     { "iscsi", "login-response", 0, 0, "idm_conn_t **", "conninfo_t **",
200     { "iscsi", "login-response", 1, 1, "iscsi_login_rsp_hdr_t **",
201     { "iscsiinfo_t **",
202     { "iscsi", "logout-command", 0, 0, "idm_conn_t **", "conninfo_t **",
203     { "iscsi", "logout-command", 1, 1, "iscsi_logout_hdr_t **",
204     { "iscsiinfo_t **",
205     { "iscsi", "logout-response", 0, 0, "idm_conn_t **", "conninfo_t **",
206     { "iscsi", "logout-response", 1, 1, "iscsi_logout_rsp_hdr_t **",
207     { "iscsiinfo_t **",
208     { "iscsi", "data-request", 0, 0, "idm_conn_t **", "conninfo_t **",
209     { "iscsi", "data-request", 1, 1, "iscsi_rtt_hdr_t **",
210     { "iscsiinfo_t **",
211     { "iscsi", "data-send", 0, 0, "idm_conn_t **", "conninfo_t **",
212     { "iscsi", "data-send", 1, 1, "iscsi_data_rsp_hdr_t **",
213     { "iscsiinfo_t **",
214     { "iscsi", "data-receive", 0, 0, "idm_conn_t **", "conninfo_t **",
215     { "iscsi", "data-receive", 1, 1, "iscsi_data_hdr_t **",
216     { "iscsiinfo_t **",
217     { "iscsi", "nop-send", 0, 0, "idm_conn_t **", "conninfo_t **",
218     { "iscsi", "nop-send", 1, 1, "iscsi_nop_in_hdr_t **", "iscsiinfo_t **",
219     { "iscsi", "nop-receive", 0, 0, "idm_conn_t **", "conninfo_t **",
220     { "iscsi", "nop-receive", 1, 1, "iscsi_nop_out_hdr_t **",
221     { "iscsiinfo_t **",
222     { "iscsi", "scsi-command", 0, 0, "idm_conn_t **", "conninfo_t **",
223     { "iscsi", "scsi-command", 1, 1, "iscsi_scsi_cmd_hdr_t **",

```

```

224     "iscsiinfo_t ** },
225     { "iscsi", "scsi-command", 2, 2, "scsi_task_t **", "scsiCmd_t ** },
226     { "iscsi", "scsi-response", 0, 0, "idm_conn_t **", "conninfo_t ** },
227     { "iscsi", "scsi-response", 1, 1, "iscsi_scsi_rsp_hdr_t **",
228       "iscsiinfo_t ** },
229     { "iscsi", "task-command", 0, 0, "idm_conn_t **", "conninfo_t ** },
230     { "iscsi", "task-command", 1, 1, "iscsi_scsi_task_mgt_hdr_t **",
231       "iscsiinfo_t ** },
232     { "iscsi", "task-response", 0, 0, "idm_conn_t **", "conninfo_t ** },
233     { "iscsi", "task-response", 1, 1, "iscsi_scsi_task_mgt_rsp_hdr_t **",
234       "iscsiinfo_t ** },
235     { "iscsi", "text-command", 0, 0, "idm_conn_t **", "conninfo_t ** },
236     { "iscsi", "text-command", 1, 1, "iscsi_text_hdr_t **",
237       "iscsiinfo_t ** },
238     { "iscsi", "text-response", 0, 0, "idm_conn_t **", "conninfo_t ** },
239     { "iscsi", "text-response", 1, 1, "iscsi_text_rsp_hdr_t **",
240       "iscsiinfo_t ** },
241     { "iscsi", "xfer-start", 0, 0, "idm_conn_t **", "conninfo_t ** },
242     { "iscsi", "xfer-start", 1, 0, "idm_conn_t **", "iscsiinfo_t ** },
243     { "iscsi", "xfer-start", 2, 1, "uintptr_t", "xferinfo_t ** },
244     { "iscsi", "xfer-start", 3, 2, "uint32_t"},
245     { "iscsi", "xfer-start", 4, 3, "uintptr_t"},
246     { "iscsi", "xfer-start", 5, 4, "uint32_t"},
247     { "iscsi", "xfer-start", 6, 5, "uint32_t"},
248     { "iscsi", "xfer-start", 7, 6, "uint32_t"},
249     { "iscsi", "xfer-start", 8, 7, "int"},
250     { "iscsi", "xfer-done", 0, 0, "idm_conn_t **", "conninfo_t ** },
251     { "iscsi", "xfer-done", 1, 0, "idm_conn_t **", "iscsiinfo_t ** },
252     { "iscsi", "xfer-done", 2, 1, "uintptr_t", "xferinfo_t ** },
253     { "iscsi", "xfer-done", 3, 2, "uint32_t"},
254     { "iscsi", "xfer-done", 4, 3, "uintptr_t"},
255     { "iscsi", "xfer-done", 5, 4, "uint32_t"},
256     { "iscsi", "xfer-done", 6, 5, "uint32_t"},
257     { "iscsi", "xfer-done", 7, 6, "uint32_t"},
258     { "iscsi", "xfer-done", 8, 7, "int"},
259
260     { "nfsv3", "op-getattr-start", 0, 0, "struct svc_req **",
261       "conninfo_t ** },
262     { "nfsv3", "op-getattr-start", 1, 1, "nfsv3oparg_t **",
263       "nfsv3opinfo_t ** },
264     { "nfsv3", "op-getattr-start", 2, 3, "GETATTR3args ** },
265     { "nfsv3", "op-getattr-done", 0, 0, "struct svc_req **",
266       "conninfo_t ** },
267     { "nfsv3", "op-getattr-done", 1, 1, "nfsv3oparg_t **",
268       "nfsv3opinfo_t ** },
269     { "nfsv3", "op-getattr-done", 2, 3, "GETATTR3res ** },
270     { "nfsv3", "op-setattr-start", 0, 0, "struct svc_req **",
271       "conninfo_t ** },
272     { "nfsv3", "op-setattr-start", 1, 1, "nfsv3oparg_t **",
273       "nfsv3opinfo_t ** },
274     { "nfsv3", "op-setattr-start", 2, 3, "SETATTR3args ** },
275     { "nfsv3", "op-setattr-done", 0, 0, "struct svc_req **",
276       "conninfo_t ** },
277     { "nfsv3", "op-setattr-done", 1, 1, "nfsv3oparg_t **",
278       "nfsv3opinfo_t ** },
279     { "nfsv3", "op-setattr-done", 2, 3, "SETATTR3res ** },
280     { "nfsv3", "op-lookup-start", 0, 0, "struct svc_req **",
281       "conninfo_t ** },
282     { "nfsv3", "op-lookup-start", 1, 1, "nfsv3oparg_t **",
283       "nfsv3opinfo_t ** },
284     { "nfsv3", "op-lookup-start", 2, 3, "LOOKUP3args ** },
285     { "nfsv3", "op-lookup-done", 0, 0, "struct svc_req **",
286       "conninfo_t ** },
287     { "nfsv3", "op-lookup-done", 1, 1, "nfsv3oparg_t **",
288       "nfsv3opinfo_t ** },
289     { "nfsv3", "op-lookup-done", 2, 3, "LOOKUP3res ** },

```

```

290     { "nfsv3", "op-access-start", 0, 0, "struct svc_req **",
291       "conninfo_t ** },
292     { "nfsv3", "op-access-start", 1, 1, "nfsv3oparg_t **",
293       "nfsv3opinfo_t ** },
294     { "nfsv3", "op-access-start", 2, 3, "ACCESS3args ** },
295     { "nfsv3", "op-access-done", 0, 0, "struct svc_req **",
296       "conninfo_t ** },
297     { "nfsv3", "op-access-done", 1, 1, "nfsv3oparg_t **",
298       "nfsv3opinfo_t ** },
299     { "nfsv3", "op-access-done", 2, 3, "ACCESS3res ** },
300     { "nfsv3", "op-commit-start", 0, 0, "struct svc_req **",
301       "conninfo_t ** },
302     { "nfsv3", "op-commit-start", 1, 1, "nfsv3oparg_t **",
303       "nfsv3opinfo_t ** },
304     { "nfsv3", "op-commit-start", 2, 3, "COMMIT3args ** },
305     { "nfsv3", "op-commit-done", 0, 0, "struct svc_req **",
306       "conninfo_t ** },
307     { "nfsv3", "op-commit-done", 1, 1, "nfsv3oparg_t **",
308       "nfsv3opinfo_t ** },
309     { "nfsv3", "op-commit-done", 2, 3, "COMMIT3res ** },
310     { "nfsv3", "op-create-start", 0, 0, "struct svc_req **",
311       "conninfo_t ** },
312     { "nfsv3", "op-create-start", 1, 1, "nfsv3oparg_t **",
313       "nfsv3opinfo_t ** },
314     { "nfsv3", "op-create-start", 2, 3, "CREATE3args ** },
315     { "nfsv3", "op-create-done", 0, 0, "struct svc_req **",
316       "conninfo_t ** },
317     { "nfsv3", "op-create-done", 1, 1, "nfsv3oparg_t **",
318       "nfsv3opinfo_t ** },
319     { "nfsv3", "op-create-done", 2, 3, "CREATE3res ** },
320     { "nfsv3", "op-fsinfo-start", 0, 0, "struct svc_req **",
321       "conninfo_t ** },
322     { "nfsv3", "op-fsinfo-start", 1, 1, "nfsv3oparg_t **",
323       "nfsv3opinfo_t ** },
324     { "nfsv3", "op-fsinfo-start", 2, 3, "FSINFO3args ** },
325     { "nfsv3", "op-fsinfo-done", 0, 0, "struct svc_req **",
326       "conninfo_t ** },
327     { "nfsv3", "op-fsinfo-done", 1, 1, "nfsv3oparg_t **",
328       "nfsv3opinfo_t ** },
329     { "nfsv3", "op-fsinfo-done", 2, 3, "FSINFO3res ** },
330     { "nfsv3", "op-fsstat-start", 0, 0, "struct svc_req **",
331       "conninfo_t ** },
332     { "nfsv3", "op-fsstat-start", 1, 1, "nfsv3oparg_t **",
333       "nfsv3opinfo_t ** },
334     { "nfsv3", "op-fsstat-start", 2, 3, "FSSTAT3args ** },
335     { "nfsv3", "op-fsstat-done", 0, 0, "struct svc_req **",
336       "conninfo_t ** },
337     { "nfsv3", "op-fsstat-done", 1, 1, "nfsv3oparg_t **",
338       "nfsv3opinfo_t ** },
339     { "nfsv3", "op-fsstat-done", 2, 3, "FSSTAT3res ** },
340     { "nfsv3", "op-link-start", 0, 0, "struct svc_req **",
341       "conninfo_t ** },
342     { "nfsv3", "op-link-start", 1, 1, "nfsv3oparg_t **",
343       "nfsv3opinfo_t ** },
344     { "nfsv3", "op-link-start", 2, 3, "LINK3args ** },
345     { "nfsv3", "op-link-done", 0, 0, "struct svc_req **",
346       "conninfo_t ** },
347     { "nfsv3", "op-link-done", 1, 1, "nfsv3oparg_t **",
348       "nfsv3opinfo_t ** },
349     { "nfsv3", "op-link-done", 2, 3, "LINK3res ** },
350     { "nfsv3", "op-mkdir-start", 0, 0, "struct svc_req **",
351       "conninfo_t ** },
352     { "nfsv3", "op-mkdir-start", 1, 1, "nfsv3oparg_t **",
353       "nfsv3opinfo_t ** },
354     { "nfsv3", "op-mkdir-start", 2, 3, "MKDIR3args ** },
355     { "nfsv3", "op-mkdir-done", 0, 0, "struct svc_req **",

```

```

356     "conninfo_t ** },
357     { "nfsv3", "op-mkdir-done", 1, 1, "nfsv3oparg_t **",
358       "nfsv3opinfo_t ** },
359     { "nfsv3", "op-mkdir-done", 2, 3, "MKDIR3res ** },
360     { "nfsv3", "op-mknod-start", 0, 0, "struct svc_req **",
361       "conninfo_t ** },
362     { "nfsv3", "op-mknod-start", 1, 1, "nfsv3oparg_t **",
363       "nfsv3opinfo_t ** },
364     { "nfsv3", "op-mknod-start", 2, 3, "MKNOD3args ** },
365     { "nfsv3", "op-mknod-done", 0, 0, "struct svc_req **",
366       "conninfo_t ** },
367     { "nfsv3", "op-mknod-done", 1, 1, "nfsv3oparg_t **",
368       "nfsv3opinfo_t ** },
369     { "nfsv3", "op-mknod-done", 2, 3, "MKNOD3res ** },
370     { "nfsv3", "op-null-start", 0, 0, "struct svc_req **",
371       "conninfo_t ** },
372     { "nfsv3", "op-null-start", 1, 1, "nfsv3oparg_t **",
373       "nfsv3opinfo_t ** },
374     { "nfsv3", "op-null-done", 0, 0, "struct svc_req **",
375       "conninfo_t ** },
376     { "nfsv3", "op-null-done", 1, 1, "nfsv3oparg_t **",
377       "nfsv3opinfo_t ** },
378     { "nfsv3", "op-pathconf-start", 0, 0, "struct svc_req **",
379       "conninfo_t ** },
380     { "nfsv3", "op-pathconf-start", 1, 1, "nfsv3oparg_t **",
381       "nfsv3opinfo_t ** },
382     { "nfsv3", "op-pathconf-start", 2, 3, "PATHCONF3args ** },
383     { "nfsv3", "op-pathconf-done", 0, 0, "struct svc_req **",
384       "conninfo_t ** },
385     { "nfsv3", "op-pathconf-done", 1, 1, "nfsv3oparg_t **",
386       "nfsv3opinfo_t ** },
387     { "nfsv3", "op-pathconf-done", 2, 3, "PATHCONF3res ** },
388     { "nfsv3", "op-read-start", 0, 0, "struct svc_req **",
389       "conninfo_t ** },
390     { "nfsv3", "op-read-start", 1, 1, "nfsv3oparg_t **",
391       "nfsv3opinfo_t ** },
392     { "nfsv3", "op-read-start", 2, 3, "READ3args ** },
393     { "nfsv3", "op-read-done", 0, 0, "struct svc_req **",
394       "conninfo_t ** },
395     { "nfsv3", "op-read-done", 1, 1, "nfsv3oparg_t **",
396       "nfsv3opinfo_t ** },
397     { "nfsv3", "op-read-done", 2, 3, "READ3res ** },
398     { "nfsv3", "op-readdir-start", 0, 0, "struct svc_req **",
399       "conninfo_t ** },
400     { "nfsv3", "op-readdir-start", 1, 1, "nfsv3oparg_t **",
401       "nfsv3opinfo_t ** },
402     { "nfsv3", "op-readdir-start", 2, 3, "READDIR3args ** },
403     { "nfsv3", "op-readdir-done", 0, 0, "struct svc_req **",
404       "conninfo_t ** },
405     { "nfsv3", "op-readdir-done", 1, 1, "nfsv3oparg_t **",
406       "nfsv3opinfo_t ** },
407     { "nfsv3", "op-readdir-done", 2, 3, "READDIR3res ** },
408     { "nfsv3", "op-readdirplus-start", 0, 0, "struct svc_req **",
409       "conninfo_t ** },
410     { "nfsv3", "op-readdirplus-start", 1, 1, "nfsv3oparg_t **",
411       "nfsv3opinfo_t ** },
412     { "nfsv3", "op-readdirplus-start", 2, 3, "READDIRPLUS3args ** },
413     { "nfsv3", "op-readdirplus-done", 0, 0, "struct svc_req **",
414       "conninfo_t ** },
415     { "nfsv3", "op-readdirplus-done", 1, 1, "nfsv3oparg_t **",
416       "nfsv3opinfo_t ** },
417     { "nfsv3", "op-readdirplus-done", 2, 3, "READDIRPLUS3res ** },
418     { "nfsv3", "op-readlink-start", 0, 0, "struct svc_req **",
419       "conninfo_t ** },
420     { "nfsv3", "op-readlink-start", 1, 1, "nfsv3oparg_t **",
421       "nfsv3opinfo_t ** },

```

```

422     { "nfsv3", "op-readlink-start", 2, 3, "READLINK3args ** },
423     { "nfsv3", "op-readlink-done", 0, 0, "struct svc_req **",
424       "conninfo_t ** },
425     { "nfsv3", "op-readlink-done", 1, 1, "nfsv3oparg_t **",
426       "nfsv3opinfo_t ** },
427     { "nfsv3", "op-readlink-done", 2, 3, "READLINK3res ** },
428     { "nfsv3", "op-remove-start", 0, 0, "struct svc_req **",
429       "conninfo_t ** },
430     { "nfsv3", "op-remove-start", 1, 1, "nfsv3oparg_t **",
431       "nfsv3opinfo_t ** },
432     { "nfsv3", "op-remove-start", 2, 3, "REMOVE3args ** },
433     { "nfsv3", "op-remove-done", 0, 0, "struct svc_req **",
434       "conninfo_t ** },
435     { "nfsv3", "op-remove-done", 1, 1, "nfsv3oparg_t **",
436       "nfsv3opinfo_t ** },
437     { "nfsv3", "op-remove-done", 2, 3, "REMOVE3res ** },
438     { "nfsv3", "op-rename-start", 0, 0, "struct svc_req **",
439       "conninfo_t ** },
440     { "nfsv3", "op-rename-start", 1, 1, "nfsv3oparg_t **",
441       "nfsv3opinfo_t ** },
442     { "nfsv3", "op-rename-start", 2, 3, "RENAME3args ** },
443     { "nfsv3", "op-rename-done", 0, 0, "struct svc_req **",
444       "conninfo_t ** },
445     { "nfsv3", "op-rename-done", 1, 1, "nfsv3oparg_t **",
446       "nfsv3opinfo_t ** },
447     { "nfsv3", "op-rename-done", 2, 3, "RENAME3res ** },
448     { "nfsv3", "op-rmdir-start", 0, 0, "struct svc_req **",
449       "conninfo_t ** },
450     { "nfsv3", "op-rmdir-start", 1, 1, "nfsv3oparg_t **",
451       "nfsv3opinfo_t ** },
452     { "nfsv3", "op-rmdir-start", 2, 3, "RMDIR3args ** },
453     { "nfsv3", "op-rmdir-done", 0, 0, "struct svc_req **",
454       "conninfo_t ** },
455     { "nfsv3", "op-rmdir-done", 1, 1, "nfsv3oparg_t **",
456       "nfsv3opinfo_t ** },
457     { "nfsv3", "op-rmdir-done", 2, 3, "RMDIR3res ** },
458     { "nfsv3", "op-setattr-start", 0, 0, "struct svc_req **",
459       "conninfo_t ** },
460     { "nfsv3", "op-setattr-start", 1, 1, "nfsv3oparg_t **",
461       "nfsv3opinfo_t ** },
462     { "nfsv3", "op-setattr-start", 2, 3, "SETATTR3args ** },
463     { "nfsv3", "op-setattr-done", 0, 0, "struct svc_req **",
464       "conninfo_t ** },
465     { "nfsv3", "op-setattr-done", 1, 1, "nfsv3oparg_t **",
466       "nfsv3opinfo_t ** },
467     { "nfsv3", "op-setattr-done", 2, 3, "SETATTR3res ** },
468     { "nfsv3", "op-symlink-start", 0, 0, "struct svc_req **",
469       "conninfo_t ** },
470     { "nfsv3", "op-symlink-start", 1, 1, "nfsv3oparg_t **",
471       "nfsv3opinfo_t ** },
472     { "nfsv3", "op-symlink-start", 2, 3, "SYMLINK3args ** },
473     { "nfsv3", "op-symlink-done", 0, 0, "struct svc_req **",
474       "conninfo_t ** },
475     { "nfsv3", "op-symlink-done", 1, 1, "nfsv3oparg_t **",
476       "nfsv3opinfo_t ** },
477     { "nfsv3", "op-symlink-done", 2, 3, "SYMLINK3res ** },
478     { "nfsv3", "op-write-start", 0, 0, "struct svc_req **",
479       "conninfo_t ** },
480     { "nfsv3", "op-write-start", 1, 1, "nfsv3oparg_t **",
481       "nfsv3opinfo_t ** },
482     { "nfsv3", "op-write-start", 2, 3, "WRITE3args ** },
483     { "nfsv3", "op-write-done", 0, 0, "struct svc_req **",
484       "conninfo_t ** },
485     { "nfsv3", "op-write-done", 1, 1, "nfsv3oparg_t **",
486       "nfsv3opinfo_t ** },
487     { "nfsv3", "op-write-done", 2, 3, "WRITE3res ** },

```

```

489 { "nfsv4", "null-start", 0, 0, "struct svc_req **", "conninfo_t **",
490 { "nfsv4", "null-done", 0, 0, "struct svc_req **", "conninfo_t **",
491 { "nfsv4", "compound-start", 0, 0, "struct compound_state **",
492   "conninfo_t **",
493 { "nfsv4", "compound-start", 1, 0, "struct compound_state **",
494   "nfsv4opinfo_t **",
495 { "nfsv4", "compound-start", 2, 1, "COMPOUND4args **",
496   "nfsv4", "compound-done", 0, 0, "struct compound_state **",
497   "conninfo_t **",
498 { "nfsv4", "compound-done", 1, 0, "struct compound_state **",
499   "nfsv4opinfo_t **",
500 { "nfsv4", "compound-done", 2, 1, "COMPOUND4res **",
501 { "nfsv4", "op-access-start", 0, 0, "struct compound_state **",
502   "conninfo_t **",
503 { "nfsv4", "op-access-start", 1, 0, "struct compound_state **",
504   "nfsv4opinfo_t **",
505 { "nfsv4", "op-access-start", 2, 1, "ACCESS4args **",
506 { "nfsv4", "op-access-done", 0, 0, "struct compound_state **",
507   "conninfo_t **",
508 { "nfsv4", "op-access-done", 1, 0, "struct compound_state **",
509   "nfsv4opinfo_t **",
510 { "nfsv4", "op-access-done", 2, 1, "ACCESS4res **",
511 { "nfsv4", "op-close-start", 0, 0, "struct compound_state **",
512   "conninfo_t **",
513 { "nfsv4", "op-close-start", 1, 0, "struct compound_state **",
514   "nfsv4opinfo_t **",
515 { "nfsv4", "op-close-start", 2, 1, "CLOSE4args **",
516 { "nfsv4", "op-close-done", 0, 0, "struct compound_state **",
517   "conninfo_t **",
518 { "nfsv4", "op-close-done", 1, 0, "struct compound_state **",
519   "nfsv4opinfo_t **",
520 { "nfsv4", "op-close-done", 2, 1, "CLOSE4res **",
521 { "nfsv4", "op-commit-start", 0, 0, "struct compound_state **",
522   "conninfo_t **",
523 { "nfsv4", "op-commit-start", 1, 0, "struct compound_state **",
524   "nfsv4opinfo_t **",
525 { "nfsv4", "op-commit-start", 2, 1, "COMMIT4args **",
526 { "nfsv4", "op-commit-done", 0, 0, "struct compound_state **",
527   "conninfo_t **",
528 { "nfsv4", "op-commit-done", 1, 0, "struct compound_state **",
529   "nfsv4opinfo_t **",
530 { "nfsv4", "op-commit-done", 2, 1, "COMMIT4res **",
531 { "nfsv4", "op-create-start", 0, 0, "struct compound_state **",
532   "conninfo_t **",
533 { "nfsv4", "op-create-start", 1, 0, "struct compound_state **",
534   "nfsv4opinfo_t **",
535 { "nfsv4", "op-create-start", 2, 1, "CREATE4args **",
536 { "nfsv4", "op-create-done", 0, 0, "struct compound_state **",
537   "conninfo_t **",
538 { "nfsv4", "op-create-done", 1, 0, "struct compound_state **",
539   "nfsv4opinfo_t **",
540 { "nfsv4", "op-create-done", 2, 1, "CREATE4res **",
541 { "nfsv4", "op-deleppurge-start", 0, 0, "struct compound_state **",
542   "conninfo_t **",
543 { "nfsv4", "op-deleppurge-start", 1, 0, "struct compound_state **",
544   "nfsv4opinfo_t **",
545 { "nfsv4", "op-deleppurge-start", 2, 1, "DELEGPURGE4args **",
546 { "nfsv4", "op-deleppurge-done", 0, 0, "struct compound_state **",
547   "conninfo_t **",
548 { "nfsv4", "op-deleppurge-done", 1, 0, "struct compound_state **",
549   "nfsv4opinfo_t **",
550 { "nfsv4", "op-deleppurge-done", 2, 1, "DELEGPURGE4res **",
551 { "nfsv4", "op-delegreturn-start", 0, 0, "struct compound_state **",
552   "conninfo_t **",
553 { "nfsv4", "op-delegreturn-start", 1, 0, "struct compound_state **",

```

```

554   "nfsv4opinfo_t **",
555 { "nfsv4", "op-delegreturn-start", 2, 1, "DELEGRETURN4args **",
556 { "nfsv4", "op-delegreturn-done", 0, 0, "struct compound_state **",
557   "conninfo_t **",
558 { "nfsv4", "op-delegreturn-done", 1, 0, "struct compound_state **",
559   "nfsv4opinfo_t **",
560 { "nfsv4", "op-delegreturn-done", 2, 1, "DELEGRETURN4res **",
561 { "nfsv4", "op-getattr-start", 0, 0, "struct compound_state **",
562   "conninfo_t **",
563 { "nfsv4", "op-getattr-start", 1, 0, "struct compound_state **",
564   "nfsv4opinfo_t **",
565 { "nfsv4", "op-getattr-start", 2, 1, "GETATTR4args **",
566 { "nfsv4", "op-getattr-done", 0, 0, "struct compound_state **",
567   "conninfo_t **",
568 { "nfsv4", "op-getattr-done", 1, 0, "struct compound_state **",
569   "nfsv4opinfo_t **",
570 { "nfsv4", "op-getattr-done", 2, 1, "GETATTR4res **",
571 { "nfsv4", "op-getfh-start", 0, 0, "struct compound_state **",
572   "conninfo_t **",
573 { "nfsv4", "op-getfh-start", 1, 0, "struct compound_state **",
574   "nfsv4opinfo_t **",
575 { "nfsv4", "op-getfh-done", 0, 0, "struct compound_state **",
576   "conninfo_t **",
577 { "nfsv4", "op-getfh-done", 1, 0, "struct compound_state **",
578   "nfsv4opinfo_t **",
579 { "nfsv4", "op-getfh-done", 2, 1, "GETFH4res **",
580 { "nfsv4", "op-link-start", 0, 0, "struct compound_state **",
581   "conninfo_t **",
582 { "nfsv4", "op-link-start", 1, 0, "struct compound_state **",
583   "nfsv4opinfo_t **",
584 { "nfsv4", "op-link-start", 2, 1, "LINK4args **",
585 { "nfsv4", "op-link-done", 0, 0, "struct compound_state **",
586   "conninfo_t **",
587 { "nfsv4", "op-link-done", 1, 0, "struct compound_state **",
588   "nfsv4opinfo_t **",
589 { "nfsv4", "op-link-done", 2, 1, "LINK4res **",
590 { "nfsv4", "op-lock-start", 0, 0, "struct compound_state **",
591   "conninfo_t **",
592 { "nfsv4", "op-lock-start", 1, 0, "struct compound_state **",
593   "nfsv4opinfo_t **",
594 { "nfsv4", "op-lock-start", 2, 1, "LOCK4args **",
595 { "nfsv4", "op-lock-done", 0, 0, "struct compound_state **",
596   "conninfo_t **",
597 { "nfsv4", "op-lock-done", 1, 0, "struct compound_state **",
598   "nfsv4opinfo_t **",
599 { "nfsv4", "op-lock-done", 2, 1, "LOCK4res **",
600 { "nfsv4", "op-lockt-start", 0, 0, "struct compound_state **",
601   "conninfo_t **",
602 { "nfsv4", "op-lockt-start", 1, 0, "struct compound_state **",
603   "nfsv4opinfo_t **",
604 { "nfsv4", "op-lockt-start", 2, 1, "LOCKT4args **",
605 { "nfsv4", "op-lockt-done", 0, 0, "struct compound_state **",
606   "conninfo_t **",
607 { "nfsv4", "op-lockt-done", 1, 0, "struct compound_state **",
608   "nfsv4opinfo_t **",
609 { "nfsv4", "op-lockt-done", 2, 1, "LOCKT4res **",
610 { "nfsv4", "op-locku-start", 0, 0, "struct compound_state **",
611   "conninfo_t **",
612 { "nfsv4", "op-locku-start", 1, 0, "struct compound_state **",
613   "nfsv4opinfo_t **",
614 { "nfsv4", "op-locku-start", 2, 1, "LOCKU4args **",
615 { "nfsv4", "op-locku-done", 0, 0, "struct compound_state **",
616   "conninfo_t **",
617 { "nfsv4", "op-locku-done", 1, 0, "struct compound_state **",
618   "nfsv4opinfo_t **",
619 { "nfsv4", "op-locku-done", 2, 1, "LOCKU4res **",

```

```

620 { "nfsv4", "op-lookup-start", 0, 0, "struct compound_state **",
621     "conninfo_t **" },
622 { "nfsv4", "op-lookup-start", 1, 0, "struct compound_state **",
623     "nfsv4opinfo_t **" },
624 { "nfsv4", "op-lookup-start", 2, 1, "LOOKUP4args **" },
625 { "nfsv4", "op-lookup-done", 0, 0, "struct compound_state **",
626     "conninfo_t **" },
627 { "nfsv4", "op-lookup-done", 1, 0, "struct compound_state **",
628     "nfsv4opinfo_t **" },
629 { "nfsv4", "op-lookup-done", 2, 1, "LOOKUP4res **" },
630 { "nfsv4", "op-lookupp-start", 0, 0, "struct compound_state **",
631     "conninfo_t **" },
632 { "nfsv4", "op-lookupp-start", 1, 0, "struct compound_state **",
633     "nfsv4opinfo_t **" },
634 { "nfsv4", "op-lookupp-done", 0, 0, "struct compound_state **",
635     "conninfo_t **" },
636 { "nfsv4", "op-lookupp-done", 1, 0, "struct compound_state **",
637     "nfsv4opinfo_t **" },
638 { "nfsv4", "op-lookupp-done", 2, 1, "LOOKUP4res **" },
639 { "nfsv4", "op-nverify-start", 0, 0, "struct compound_state **",
640     "conninfo_t **" },
641 { "nfsv4", "op-nverify-start", 1, 0, "struct compound_state **",
642     "nfsv4opinfo_t **" },
643 { "nfsv4", "op-nverify-start", 2, 1, "NVERIFY4args **" },
644 { "nfsv4", "op-nverify-done", 0, 0, "struct compound_state **",
645     "conninfo_t **" },
646 { "nfsv4", "op-nverify-done", 1, 0, "struct compound_state **",
647     "nfsv4opinfo_t **" },
648 { "nfsv4", "op-nverify-done", 2, 1, "NVERIFY4res **" },
649 { "nfsv4", "op-open-start", 0, 0, "struct compound_state **",
650     "conninfo_t **" },
651 { "nfsv4", "op-open-start", 1, 0, "struct compound_state **",
652     "nfsv4opinfo_t **" },
653 { "nfsv4", "op-open-start", 2, 1, "OPEN4args **" },
654 { "nfsv4", "op-open-done", 0, 0, "struct compound_state **",
655     "conninfo_t **" },
656 { "nfsv4", "op-open-done", 1, 0, "struct compound_state **",
657     "nfsv4opinfo_t **" },
658 { "nfsv4", "op-open-done", 2, 1, "OPEN4res **" },
659 { "nfsv4", "op-open-confirm-start", 0, 0, "struct compound_state **",
660     "conninfo_t **" },
661 { "nfsv4", "op-open-confirm-start", 1, 0, "struct compound_state **",
662     "nfsv4opinfo_t **" },
663 { "nfsv4", "op-open-confirm-start", 2, 1, "OPEN_CONFIRM4args **" },
664 { "nfsv4", "op-open-confirm-done", 0, 0, "struct compound_state **",
665     "conninfo_t **" },
666 { "nfsv4", "op-open-confirm-done", 1, 0, "struct compound_state **",
667     "nfsv4opinfo_t **" },
668 { "nfsv4", "op-open-confirm-done", 2, 1, "OPEN_CONFIRM4res **" },
669 { "nfsv4", "op-open-downgrade-start", 0, 0, "struct compound_state **",
670     "conninfo_t **" },
671 { "nfsv4", "op-open-downgrade-start", 1, 0, "struct compound_state **",
672     "nfsv4opinfo_t **" },
673 { "nfsv4", "op-open-downgrade-start", 2, 1, "OPEN_DOWNGRADE4args **" },
674 { "nfsv4", "op-open-downgrade-done", 0, 0, "struct compound_state **",
675     "conninfo_t **" },
676 { "nfsv4", "op-open-downgrade-done", 1, 0, "struct compound_state **",
677     "nfsv4opinfo_t **" },
678 { "nfsv4", "op-open-downgrade-done", 2, 1, "OPEN_DOWNGRADE4res **" },
679 { "nfsv4", "op-openattr-start", 0, 0, "struct compound_state **",
680     "conninfo_t **" },
681 { "nfsv4", "op-openattr-start", 1, 0, "struct compound_state **",
682     "nfsv4opinfo_t **" },
683 { "nfsv4", "op-openattr-start", 2, 1, "OPENATTR4args **" },
684 { "nfsv4", "op-openattr-done", 0, 0, "struct compound_state **",
685     "conninfo_t **" },

```

```

686 { "nfsv4", "op-openattr-done", 1, 0, "struct compound_state **",
687     "nfsv4opinfo_t **" },
688 { "nfsv4", "op-openattr-done", 2, 1, "OPENATTR4res **" },
689 { "nfsv4", "op-putfh-start", 0, 0, "struct compound_state **",
690     "conninfo_t **" },
691 { "nfsv4", "op-putfh-start", 1, 0, "struct compound_state **",
692     "nfsv4opinfo_t **" },
693 { "nfsv4", "op-putfh-start", 2, 1, "PUTFH4args **" },
694 { "nfsv4", "op-putfh-done", 0, 0, "struct compound_state **",
695     "conninfo_t **" },
696 { "nfsv4", "op-putfh-done", 1, 0, "struct compound_state **",
697     "nfsv4opinfo_t **" },
698 { "nfsv4", "op-putfh-done", 2, 1, "PUTFH4res **" },
699 { "nfsv4", "op-putpubfh-start", 0, 0, "struct compound_state **",
700     "conninfo_t **" },
701 { "nfsv4", "op-putpubfh-start", 1, 0, "struct compound_state **",
702     "nfsv4opinfo_t **" },
703 { "nfsv4", "op-putpubfh-done", 0, 0, "struct compound_state **",
704     "conninfo_t **" },
705 { "nfsv4", "op-putpubfh-done", 1, 0, "struct compound_state **",
706     "nfsv4opinfo_t **" },
707 { "nfsv4", "op-putpubfh-done", 2, 1, "PUTPUBFH4res **" },
708 { "nfsv4", "op-putrootfh-start", 0, 0, "struct compound_state **",
709     "conninfo_t **" },
710 { "nfsv4", "op-putrootfh-start", 1, 0, "struct compound_state **",
711     "nfsv4opinfo_t **" },
712 { "nfsv4", "op-putrootfh-done", 0, 0, "struct compound_state **",
713     "conninfo_t **" },
714 { "nfsv4", "op-putrootfh-done", 1, 0, "struct compound_state **",
715     "nfsv4opinfo_t **" },
716 { "nfsv4", "op-putrootfh-done", 2, 1, "PUTROOTFH4res **" },
717 { "nfsv4", "op-read-start", 0, 0, "struct compound_state **",
718     "conninfo_t **" },
719 { "nfsv4", "op-read-start", 1, 0, "struct compound_state **",
720     "nfsv4opinfo_t **" },
721 { "nfsv4", "op-read-start", 2, 1, "READ4args **" },
722 { "nfsv4", "op-read-done", 0, 0, "struct compound_state **",
723     "conninfo_t **" },
724 { "nfsv4", "op-read-done", 1, 0, "struct compound_state **",
725     "nfsv4opinfo_t **" },
726 { "nfsv4", "op-read-done", 2, 1, "READ4res **" },
727 { "nfsv4", "op-readlink-start", 0, 0, "struct compound_state **",
728     "conninfo_t **" },
729 { "nfsv4", "op-readlink-start", 1, 0, "struct compound_state **",
730     "nfsv4opinfo_t **" },
731 { "nfsv4", "op-readlink-start", 2, 1, "READLINK4args **" },
732 { "nfsv4", "op-readlink-done", 0, 0, "struct compound_state **",
733     "conninfo_t **" },
734 { "nfsv4", "op-readlink-done", 1, 0, "struct compound_state **",
735     "nfsv4opinfo_t **" },
736 { "nfsv4", "op-readlink-done", 2, 1, "READLINK4res **" },
737 { "nfsv4", "op-readlink-start", 0, 0, "struct compound_state **",
738     "conninfo_t **" },
739 { "nfsv4", "op-readlink-start", 1, 0, "struct compound_state **",
740     "nfsv4opinfo_t **" },
741 { "nfsv4", "op-readlink-done", 0, 0, "struct compound_state **",
742     "conninfo_t **" },
743 { "nfsv4", "op-readlink-done", 1, 0, "struct compound_state **",
744     "nfsv4opinfo_t **" },
745 { "nfsv4", "op-readlink-done", 2, 1, "READLINK4res **" },
746 { "nfsv4", "op-release-lockowner-start", 0, 0,
747     "struct compound_state **", "conninfo_t **" },
748 { "nfsv4", "op-release-lockowner-start", 1, 0,
749     "struct compound_state **", "nfsv4opinfo_t **" },
750 { "nfsv4", "op-release-lockowner-start", 2, 1,
751     "RELEASE_LOCKOWNER4args **" },

```

```

752 { "nfsv4", "op-release-lockowner-done", 0, 0,
753     "struct compound_state **", "conninfo_t " },
754 { "nfsv4", "op-release-lockowner-done", 1, 0,
755     "struct compound_state **", "nfsv4opinfo_t " },
756 { "nfsv4", "op-release-lockowner-done", 2, 1,
757     "RELEASE_LOCKOWNER4res " },
758 { "nfsv4", "op-remove-start", 0, 0, "struct compound_state **",
759     "conninfo_t " },
760 { "nfsv4", "op-remove-start", 1, 0, "struct compound_state **",
761     "nfsv4opinfo_t " },
762 { "nfsv4", "op-remove-start", 2, 1, "REMOVE4args " },
763 { "nfsv4", "op-remove-done", 0, 0, "struct compound_state **",
764     "conninfo_t " },
765 { "nfsv4", "op-remove-done", 1, 0, "struct compound_state **",
766     "nfsv4opinfo_t " },
767 { "nfsv4", "op-remove-done", 2, 1, "REMOVE4res " },
768 { "nfsv4", "op-rename-start", 0, 0, "struct compound_state **",
769     "conninfo_t " },
770 { "nfsv4", "op-rename-start", 1, 0, "struct compound_state **",
771     "nfsv4opinfo_t " },
772 { "nfsv4", "op-rename-start", 2, 1, "RENAME4args " },
773 { "nfsv4", "op-rename-done", 0, 0, "struct compound_state **",
774     "conninfo_t " },
775 { "nfsv4", "op-rename-done", 1, 0, "struct compound_state **",
776     "nfsv4opinfo_t " },
777 { "nfsv4", "op-rename-done", 2, 1, "RENAME4res " },
778 { "nfsv4", "op-renew-start", 0, 0, "struct compound_state **",
779     "conninfo_t " },
780 { "nfsv4", "op-renew-start", 1, 0, "struct compound_state **",
781     "nfsv4opinfo_t " },
782 { "nfsv4", "op-renew-start", 2, 1, "RENEW4args " },
783 { "nfsv4", "op-renew-done", 0, 0, "struct compound_state **",
784     "conninfo_t " },
785 { "nfsv4", "op-renew-done", 1, 0, "struct compound_state **",
786     "nfsv4opinfo_t " },
787 { "nfsv4", "op-renew-done", 2, 1, "RENEW4res " },
788 { "nfsv4", "op-restorefh-start", 0, 0, "struct compound_state **",
789     "conninfo_t " },
790 { "nfsv4", "op-restorefh-start", 1, 0, "struct compound_state **",
791     "nfsv4opinfo_t " },
792 { "nfsv4", "op-restorefh-done", 0, 0, "struct compound_state **",
793     "conninfo_t " },
794 { "nfsv4", "op-restorefh-done", 1, 0, "struct compound_state **",
795     "nfsv4opinfo_t " },
796 { "nfsv4", "op-restorefh-done", 2, 1, "RESTOREFH4res " },
797 { "nfsv4", "op-savefh-start", 0, 0, "struct compound_state **",
798     "conninfo_t " },
799 { "nfsv4", "op-savefh-start", 1, 0, "struct compound_state **",
800     "nfsv4opinfo_t " },
801 { "nfsv4", "op-savefh-done", 0, 0, "struct compound_state **",
802     "conninfo_t " },
803 { "nfsv4", "op-savefh-done", 1, 0, "struct compound_state **",
804     "nfsv4opinfo_t " },
805 { "nfsv4", "op-savefh-done", 2, 1, "SAVEFH4res " },
806 { "nfsv4", "op-secinfo-start", 0, 0, "struct compound_state **",
807     "conninfo_t " },
808 { "nfsv4", "op-secinfo-start", 1, 0, "struct compound_state **",
809     "nfsv4opinfo_t " },
810 { "nfsv4", "op-secinfo-start", 2, 1, "SECINFO4args " },
811 { "nfsv4", "op-secinfo-done", 0, 0, "struct compound_state **",
812     "conninfo_t " },
813 { "nfsv4", "op-secinfo-done", 1, 0, "struct compound_state **",
814     "nfsv4opinfo_t " },
815 { "nfsv4", "op-secinfo-done", 2, 1, "SECINFO4res " },
816 { "nfsv4", "op-setattr-start", 0, 0, "struct compound_state **",
817     "conninfo_t " },

```

```

818 { "nfsv4", "op-setattr-start", 1, 0, "struct compound_state **",
819     "nfsv4opinfo_t " },
820 { "nfsv4", "op-setattr-start", 2, 1, "SETATTR4args " },
821 { "nfsv4", "op-setattr-done", 0, 0, "struct compound_state **",
822     "conninfo_t " },
823 { "nfsv4", "op-setattr-done", 1, 0, "struct compound_state **",
824     "nfsv4opinfo_t " },
825 { "nfsv4", "op-setattr-done", 2, 1, "SETATTR4res " },
826 { "nfsv4", "op-setclientid-start", 0, 0, "struct compound_state **",
827     "conninfo_t " },
828 { "nfsv4", "op-setclientid-start", 1, 0, "struct compound_state **",
829     "nfsv4opinfo_t " },
830 { "nfsv4", "op-setclientid-start", 2, 1, "SETCLIENTID4args " },
831 { "nfsv4", "op-setclientid-done", 0, 0, "struct compound_state **",
832     "conninfo_t " },
833 { "nfsv4", "op-setclientid-done", 1, 0, "struct compound_state **",
834     "nfsv4opinfo_t " },
835 { "nfsv4", "op-setclientid-done", 2, 1, "SETCLIENTID4res " },
836 { "nfsv4", "op-setclientid-confirm-start", 0, 0,
837     "struct compound_state **", "conninfo_t " },
838 { "nfsv4", "op-setclientid-confirm-start", 1, 0,
839     "struct compound_state **", "nfsv4opinfo_t " },
840 { "nfsv4", "op-setclientid-confirm-start", 2, 1,
841     "SETCLIENTID_CONFIRM4args " },
842 { "nfsv4", "op-setclientid-confirm-done", 0, 0,
843     "struct compound_state **", "conninfo_t " },
844 { "nfsv4", "op-setclientid-confirm-done", 1, 0,
845     "struct compound_state **", "nfsv4opinfo_t " },
846 { "nfsv4", "op-setclientid-confirm-done", 2, 1,
847     "SETCLIENTID_CONFIRM4res " },
848 { "nfsv4", "op-verify-start", 0, 0, "struct compound_state **",
849     "conninfo_t " },
850 { "nfsv4", "op-verify-start", 1, 0, "struct compound_state **",
851     "nfsv4opinfo_t " },
852 { "nfsv4", "op-verify-start", 2, 1, "VERIFY4args " },
853 { "nfsv4", "op-verify-done", 0, 0, "struct compound_state **",
854     "conninfo_t " },
855 { "nfsv4", "op-verify-done", 1, 0, "struct compound_state **",
856     "nfsv4opinfo_t " },
857 { "nfsv4", "op-verify-done", 2, 1, "VERIFY4res " },
858 { "nfsv4", "op-write-start", 0, 0, "struct compound_state **",
859     "conninfo_t " },
860 { "nfsv4", "op-write-start", 1, 0, "struct compound_state **",
861     "nfsv4opinfo_t " },
862 { "nfsv4", "op-write-start", 2, 1, "WRITE4args " },
863 { "nfsv4", "op-write-done", 0, 0, "struct compound_state **",
864     "conninfo_t " },
865 { "nfsv4", "op-write-done", 1, 0, "struct compound_state **",
866     "nfsv4opinfo_t " },
867 { "nfsv4", "op-write-done", 2, 1, "WRITE4res " },
868 { "nfsv4", "cb-recall-start", 0, 0, "rfs4_client_t **",
869     "conninfo_t " },
870 { "nfsv4", "cb-recall-start", 1, 1, "rfs4_deleg_state_t **",
871     "nfsv4cbinfo_t " },
872 { "nfsv4", "cb-recall-start", 2, 2, "CB_RECALL4args " },
873 { "nfsv4", "cb-recall-done", 0, 0, "rfs4_client_t **",
874     "conninfo_t " },
875 { "nfsv4", "cb-recall-done", 1, 1, "rfs4_deleg_state_t **",
876     "nfsv4cbinfo_t " },
877 { "nfsv4", "cb-recall-done", 2, 2, "CB_RECALL4res " },
878
879 { "ip", "send", 0, 0, "mbk_t **", "pktinfo_t " },
880 { "ip", "send", 1, 1, "conn_t **", "csinfo_t " },
881 { "ip", "send", 2, 2, "void_ip_t **", "ipinfo_t " },
882 { "ip", "send", 3, 3, "dtrace_ipsr_ill_t **", "ifinfo_t " },
883 { "ip", "send", 4, 4, "ipha_t **", "ipv4info_t " },

```

```

884 { "ip", "send", 5, 5, "ip6_t **", "ip6info_t **" },
885 { "ip", "send", 6, 6, "int" }, /* used by _dtrace_ipsr_ill_t */
886 { "ip", "receive", 0, 0, "mbk_t **", "pktinfo_t **" },
887 { "ip", "receive", 1, 1, "conn_t **", "csinfo_t **" },
888 { "ip", "receive", 2, 2, "void_ip_t **", "ipinfo_t **" },
889 { "ip", "receive", 3, 3, "_dtrace_ipsr_ill_t **", "ifinfo_t **" },
890 { "ip", "receive", 4, 4, "ipha_t **", "ip4info_t **" },
891 { "ip", "receive", 5, 5, "ip6_t **", "ip6info_t **" },
892 { "ip", "receive", 6, 6, "int" }, /* used by _dtrace_ipsr_ill_t */

894 { "tcp", "connect-established", 0, 0, "mbk_t **", "pktinfo_t **" },
895 { "tcp", "connect-established", 1, 1, "ip_xmit_attr_t **",
896   "csinfo_t **" },
897 { "tcp", "connect-established", 2, 2, "void_ip_t **", "ipinfo_t **" },
898 { "tcp", "connect-established", 3, 3, "tcp_t **", "tcpsinfo_t **" },
899 { "tcp", "connect-established", 4, 4, "tcph_t **", "tcpinfo_t **" },
900 { "tcp", "connect-refused", 0, 0, "mbk_t **", "pktinfo_t **" },
901 { "tcp", "connect-refused", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
902 { "tcp", "connect-refused", 2, 2, "void_ip_t **", "ipinfo_t **" },
903 { "tcp", "connect-refused", 3, 3, "tcp_t **", "tcpsinfo_t **" },
904 { "tcp", "connect-refused", 4, 4, "tcph_t **", "tcpinfo_t **" },
905 { "tcp", "connect-request", 0, 0, "mbk_t **", "pktinfo_t **" },
906 { "tcp", "connect-request", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
907 { "tcp", "connect-request", 2, 2, "void_ip_t **", "ipinfo_t **" },
908 { "tcp", "connect-request", 3, 3, "tcp_t **", "tcpsinfo_t **" },
909 { "tcp", "connect-request", 4, 4, "tcph_t **", "tcpinfo_t **" },
910 { "tcp", "accept-established", 0, 0, "mbk_t **", "pktinfo_t **" },
911 { "tcp", "accept-established", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
912 { "tcp", "accept-established", 2, 2, "void_ip_t **", "ipinfo_t **" },
913 { "tcp", "accept-established", 3, 3, "tcp_t **", "tcpsinfo_t **" },
914 { "tcp", "accept-established", 4, 4, "tcph_t **", "tcpinfo_t **" },
915 { "tcp", "accept-refused", 0, 0, "mbk_t **", "pktinfo_t **" },
916 { "tcp", "accept-refused", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
917 { "tcp", "accept-refused", 2, 2, "void_ip_t **", "ipinfo_t **" },
918 { "tcp", "accept-refused", 3, 3, "tcp_t **", "tcpsinfo_t **" },
919 { "tcp", "accept-refused", 4, 4, "tcph_t **", "tcpinfo_t **" },
920 { "tcp", "state-change", 0, 0, "void", "void" },
921 { "tcp", "state-change", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
922 { "tcp", "state-change", 2, 2, "void", "void" },
923 { "tcp", "state-change", 3, 3, "tcp_t **", "tcpsinfo_t **" },
924 { "tcp", "state-change", 4, 4, "void", "void" },
925 { "tcp", "state-change", 5, 5, "int32_t", "tcplinfo_t **" },
926 { "tcp", "send", 0, 0, "mbk_t **", "pktinfo_t **" },
927 { "tcp", "send", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
928 { "tcp", "send", 2, 2, "_dtrace_tcp_void_ip_t **", "ipinfo_t **" },
929 { "tcp", "send", 3, 3, "tcp_t **", "tcpsinfo_t **" },
930 { "tcp", "send", 4, 4, "_dtrace_tcp_tcph_t **", "tcpinfo_t **" },
931 { "tcp", "receive", 0, 0, "mbk_t **", "pktinfo_t **" },
932 { "tcp", "receive", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
933 { "tcp", "receive", 2, 2, "_dtrace_tcp_void_ip_t **", "ipinfo_t **" },
934 { "tcp", "receive", 3, 3, "tcp_t **", "tcpsinfo_t **" },
935 { "tcp", "receive", 4, 4, "_dtrace_tcp_tcph_t **", "tcpinfo_t **" },

937 { "udp", "send", 0, 0, "mbk_t **", "pktinfo_t **" },
938 { "udp", "send", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
939 { "udp", "send", 2, 2, "void_ip_t **", "ipinfo_t **" },
940 { "udp", "send", 3, 3, "udp_t **", "udpsinfo_t **" },
941 { "udp", "send", 4, 4, "udpha_t **", "udpinfo_t **" },
942 { "udp", "receive", 0, 0, "mbk_t **", "pktinfo_t **" },
943 { "udp", "receive", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
944 { "udp", "receive", 2, 2, "void_ip_t **", "ipinfo_t **" },
945 { "udp", "receive", 3, 3, "udp_t **", "udpsinfo_t **" },
946 { "udp", "receive", 4, 4, "udpha_t **", "udpinfo_t **" },

948 { "dccp", "connect-established", 0, 0, "mbk_t **", "pktinfo_t **" },
949 { "dccp", "connect-established", 1, 1, "ip_xmit_attr_t **",

```

```

950   "csinfo_t **" },
951 { "dccp", "connect-established", 2, 2, "void_ip_t **", "ipinfo_t **" },
952 { "dccp", "connect-established", 3, 3, "dccp_t **", "dccpsinfo_t **" },
953 { "dccp", "connect-established", 4, 4, "dccph_t **", "dccpinfo_t **" },
954 { "dccp", "connect-refused", 0, 0, "mbk_t **", "pktinfo_t **" },
955 { "dccp", "connect-refused", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
956 { "dccp", "connect-refused", 2, 2, "void_ip_t **", "ipinfo_t **" },
957 { "dccp", "connect-refused", 3, 3, "dccp_t **", "dccpsinfo_t **" },
958 { "dccp", "connect-refused", 4, 4, "dccph_t **", "dccpinfo_t **" },
959 { "dccp", "connect-request", 0, 0, "mbk_t **", "pktinfo_t **" },
960 { "dccp", "connect-request", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
961 { "dccp", "connect-request", 2, 2, "void_ip_t **", "ipinfo_t **" },
962 { "dccp", "connect-request", 3, 3, "dccp_t **", "dccpsinfo_t **" },
963 { "dccp", "connect-request", 4, 4, "dccph_t **", "dccpinfo_t **" },
964 { "dccp", "accept-established", 0, 0, "mbk_t **", "pktinfo_t **" },
965 { "dccp", "accept-established", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
966 { "dccp", "accept-established", 2, 2, "void_ip_t **", "ipinfo_t **" },
967 { "dccp", "accept-established", 3, 3, "dccp_t **", "dccpsinfo_t **" },
968 { "dccp", "accept-established", 4, 4, "dccph_t **", "dccpinfo_t **" },
969 { "dccp", "accept-refused", 0, 0, "mbk_t **", "pktinfo_t **" },
970 { "dccp", "accept-refused", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
971 { "dccp", "accept-refused", 2, 2, "void_ip_t **", "ipinfo_t **" },
972 { "dccp", "accept-refused", 3, 3, "dccp_t **", "dccpsinfo_t **" },
973 { "dccp", "accept-refused", 4, 4, "dccph_t **", "dccpinfo_t **" },
974 { "dccp", "state-change", 0, 0, "void", "void" },
975 { "dccp", "state-change", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
976 { "dccp", "state-change", 2, 2, "void", "void" },
977 { "dccp", "state-change", 3, 3, "dccp_t **", "dccpsinfo_t **" },
978 { "dccp", "state-change", 4, 4, "void", "void" },
979 { "dccp", "state-change", 5, 5, "int32_t", "dccplinfo_t **" },
980 { "dccp", "send", 0, 0, "mbk_t **", "pktinfo_t **" },
981 { "dccp", "send", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
982 { "dccp", "send", 2, 2, "_dtrace_dccp_void_ip_t **", "ipinfo_t **" },
983 { "dccp", "send", 3, 3, "dccp_t **", "dccpsinfo_t **" },
984 { "dccp", "send", 4, 4, "_dtrace_dccp_dccph_t **", "dccpinfo_t **" },
985 { "dccp", "receive", 0, 0, "mbk_t **", "pktinfo_t **" },
986 { "dccp", "receive", 1, 1, "ip_xmit_attr_t **", "csinfo_t **" },
987 { "dccp", "receive", 2, 2, "_dtrace_dccp_void_ip_t **", "ipinfo_t **" },
988 { "dccp", "receive", 3, 3, "dccp_t **", "dccpsinfo_t **" },
989 { "dccp", "receive", 4, 4, "_dtrace_dccp_dccph_t **", "dccpinfo_t **" },

991 #endif /* ! codereview */
992 { "sysevent", "post", 0, 0, "evch_bind_t **", "syseventchaninfo_t **" },
993 { "sysevent", "post", 1, 1, "sysevent_impl_t **", "syseventinfo_t **" },

995 { "xpv", "add-to-physmap-end", 0, 0, "int" },
996 { "xpv", "add-to-physmap-start", 0, 0, "domid_t" },
997 { "xpv", "add-to-physmap-start", 1, 1, "uint_t" },
998 { "xpv", "add-to-physmap-start", 2, 2, "ulong_t" },
999 { "xpv", "add-to-physmap-start", 3, 3, "ulong_t" },
1000 { "xpv", "decrease-reservation-end", 0, 0, "int" },
1001 { "xpv", "decrease-reservation-start", 0, 0, "domid_t" },
1002 { "xpv", "decrease-reservation-start", 1, 1, "ulong_t" },
1003 { "xpv", "decrease-reservation-start", 2, 2, "uint_t" },
1004 { "xpv", "decrease-reservation-start", 3, 3, "ulong_t" },
1005 { "xpv", "dom-create-start", 0, 0, "xen_domctl_t **" },
1006 { "xpv", "dom-destroy-start", 0, 0, "domid_t" },
1007 { "xpv", "dom-pause-start", 0, 0, "domid_t" },
1008 { "xpv", "dom-unpause-start", 0, 0, "domid_t" },
1009 { "xpv", "dom-create-end", 0, 0, "int" },
1010 { "xpv", "dom-destroy-end", 0, 0, "int" },
1011 { "xpv", "dom-pause-end", 0, 0, "int" },
1012 { "xpv", "dom-unpause-end", 0, 0, "int" },
1013 { "xpv", "evtchn-op-end", 0, 0, "int" },
1014 { "xpv", "evtchn-op-start", 0, 0, "int" },
1015 { "xpv", "evtchn-op-start", 1, 1, "void **" },

```



```

1016 { "xpv", "increase-reservation-end", 0, 0, "int" },
1017 { "xpv", "increase-reservation-start", 0, 0, "domid_t" },
1018 { "xpv", "increase-reservation-start", 1, 1, "ulong_t" },
1019 { "xpv", "increase-reservation-start", 2, 2, "uint_t" },
1020 { "xpv", "increase-reservation-start", 3, 3, "ulong_t" },
1021 { "xpv", "mmap-end", 0, 0, "int" },
1022 { "xpv", "mmap-entry", 0, 0, "ulong_t" },
1023 { "xpv", "mmap-entry", 1, 1, "ulong_t" },
1024 { "xpv", "mmap-entry", 2, 2, "ulong_t" },
1025 { "xpv", "mmap-start", 0, 0, "domid_t" },
1026 { "xpv", "mmap-start", 1, 1, "int" },
1027 { "xpv", "mmap-start", 2, 2, "privcmd_mmap_entry_t" },
1028 { "xpv", "mmapbatch-end", 0, 0, "int" },
1029 { "xpv", "mmapbatch-end", 1, 1, "struct seg" },
1030 { "xpv", "mmapbatch-end", 2, 2, "caddr_t" },
1031 { "xpv", "mmapbatch-start", 0, 0, "domid_t" },
1032 { "xpv", "mmapbatch-start", 1, 1, "int" },
1033 { "xpv", "mmapbatch-start", 2, 2, "caddr_t" },
1034 { "xpv", "mmu-ext-op-end", 0, 0, "int" },
1035 { "xpv", "mmu-ext-op-start", 0, 0, "int" },
1036 { "xpv", "mmu-ext-op-start", 1, 1, "struct mmuext_op" },
1037 { "xpv", "mmu-update-start", 0, 0, "int" },
1038 { "xpv", "mmu-update-start", 1, 1, "int" },
1039 { "xpv", "mmu-update-start", 2, 2, "mmu_update_t" },
1040 { "xpv", "mmu-update-end", 0, 0, "int" },
1041 { "xpv", "populate-physmap-end", 0, 0, "int" },
1042 { "xpv", "populate-physmap-start", 0, 0, "domid_t" },
1043 { "xpv", "populate-physmap-start", 1, 1, "ulong_t" },
1044 { "xpv", "populate-physmap-start", 2, 2, "ulong_t" },
1045 { "xpv", "set-memory-map-end", 0, 0, "int" },
1046 { "xpv", "set-memory-map-start", 0, 0, "domid_t" },
1047 { "xpv", "set-memory-map-start", 1, 1, "int" },
1048 { "xpv", "set-memory-map-start", 2, 2, "struct xen_memory_map" },
1049 { "xpv", "setvcpucontext-end", 0, 0, "int" },
1050 { "xpv", "setvcpucontext-start", 0, 0, "domid_t" },
1051 { "xpv", "setvcpucontext-start", 1, 1, "vcpu_guest_context_t" },

1053 { "srp", "service-up", 0, 0, "srpt_session_t", "conninfo_t" },
1054 { "srp", "service-up", 1, 0, "srpt_session_t", "srp_portinfo_t" },
1055 { "srp", "service-down", 0, 0, "srpt_session_t", "conninfo_t" },
1056 { "srp", "service-down", 1, 0, "srpt_session_t",
1057   "srp_portinfo_t" },
1058 { "srp", "login-command", 0, 0, "srpt_session_t", "conninfo_t" },
1059 { "srp", "login-command", 1, 0, "srpt_session_t",
1060   "srp_portinfo_t" },
1061 { "srp", "login-command", 2, 1, "srp_login_req_t",
1062   "srp_logininfo_t" },
1063 { "srp", "login-response", 0, 0, "srpt_session_t", "conninfo_t" },
1064 { "srp", "login-response", 1, 0, "srpt_session_t",
1065   "srp_portinfo_t" },
1066 { "srp", "login-response", 2, 1, "srp_login_rsp_t",
1067   "srp_logininfo_t" },
1068 { "srp", "login-response", 3, 2, "srp_login_rej_t" },
1069 { "srp", "logout-command", 0, 0, "srpt_channel_t", "conninfo_t" },
1070 { "srp", "logout-command", 1, 0, "srpt_channel_t",
1071   "srp_portinfo_t" },
1072 { "srp", "task-command", 0, 0, "srpt_channel_t", "conninfo_t" },
1073 { "srp", "task-command", 1, 0, "srpt_channel_t",
1074   "srp_portinfo_t" },
1075 { "srp", "task-command", 2, 1, "srp_cmd_req_t", "srp_taskinfo_t" },
1076 { "srp", "task-response", 0, 0, "srpt_channel_t", "conninfo_t" },
1077 { "srp", "task-response", 1, 0, "srpt_channel_t",
1078   "srp_portinfo_t" },
1079 { "srp", "task-response", 2, 1, "srp_rsp_t", "srp_taskinfo_t" },
1080 { "srp", "task-response", 3, 2, "scsi_task_t" },
1081 { "srp", "task-response", 4, 3, "int8_t" },

```

```

1082 { "srp", "scsi-command", 0, 0, "srpt_channel_t", "conninfo_t" },
1083 { "srp", "scsi-command", 1, 0, "srpt_channel_t",
1084   "srp_portinfo_t" },
1085 { "srp", "scsi-command", 2, 1, "scsi_task_t", "scsicmd_t" },
1086 { "srp", "scsi-command", 3, 2, "srp_cmd_req_t", "srp_taskinfo_t" },
1087 { "srp", "scsi-response", 0, 0, "srpt_channel_t", "conninfo_t" },
1088 { "srp", "scsi-response", 1, 0, "srpt_channel_t",
1089   "srp_portinfo_t" },
1090 { "srp", "scsi-response", 2, 1, "srp_rsp_t", "srp_taskinfo_t" },
1091 { "srp", "scsi-response", 3, 2, "scsi_task_t" },
1092 { "srp", "scsi-response", 4, 3, "int8_t" },
1093 { "srp", "xfer-start", 0, 0, "srpt_channel_t", "conninfo_t" },
1094 { "srp", "xfer-start", 1, 0, "srpt_channel_t",
1095   "srp_portinfo_t" },
1096 { "srp", "xfer-start", 2, 1, "ibt_wr_ds_t", "xferinfo_t" },
1097 { "srp", "xfer-start", 3, 2, "srpt_iu_t", "srp_taskinfo_t" },
1098 { "srp", "xfer-start", 4, 3, "ibt_send_wr_t" },
1099 { "srp", "xfer-start", 5, 4, "uint32_t" },
1100 { "srp", "xfer-start", 6, 5, "uint32_t" },
1101 { "srp", "xfer-start", 7, 6, "uint32_t" },
1102 { "srp", "xfer-start", 8, 7, "uint32_t" },
1103 { "srp", "xfer-done", 0, 0, "srpt_channel_t", "conninfo_t" },
1104 { "srp", "xfer-done", 1, 0, "srpt_channel_t",
1105   "srp_portinfo_t" },
1106 { "srp", "xfer-done", 2, 1, "ibt_wr_ds_t", "xferinfo_t" },
1107 { "srp", "xfer-done", 3, 2, "srpt_iu_t", "srp_taskinfo_t" },
1108 { "srp", "xfer-done", 4, 3, "ibt_send_wr_t" },
1109 { "srp", "xfer-done", 5, 4, "uint32_t" },
1110 { "srp", "xfer-done", 6, 5, "uint32_t" },
1111 { "srp", "xfer-done", 7, 6, "uint32_t" },
1112 { "srp", "xfer-done", 8, 7, "uint32_t" },

1114 { "fc", "link-up", 0, 0, "fct_i_local_port_t", "conninfo_t" },
1115 { "fc", "link-down", 0, 0, "fct_i_local_port_t", "conninfo_t" },
1116 { "fc", "fabric-login-start", 0, 0, "fct_i_local_port_t",
1117   "conninfo_t" },
1118 { "fc", "fabric-login-start", 1, 0, "fct_i_local_port_t",
1119   "fc_port_info_t" },
1120 { "fc", "fabric-login-end", 0, 0, "fct_i_local_port_t",
1121   "conninfo_t" },
1122 { "fc", "fabric-login-end", 1, 0, "fct_i_local_port_t",
1123   "fc_port_info_t" },
1124 { "fc", "rport-login-start", 0, 0, "fct_cmd_t",
1125   "conninfo_t" },
1126 { "fc", "rport-login-start", 1, 1, "fct_local_port_t",
1127   "fc_port_info_t" },
1128 { "fc", "rport-login-start", 2, 2, "fct_i_remote_port_t",
1129   "fc_port_info_t" },
1130 { "fc", "rport-login-start", 3, 3, "int", "int" },
1131 { "fc", "rport-login-end", 0, 0, "fct_cmd_t",
1132   "conninfo_t" },
1133 { "fc", "rport-login-end", 1, 1, "fct_local_port_t",
1134   "fc_port_info_t" },
1135 { "fc", "rport-login-end", 2, 2, "fct_i_remote_port_t",
1136   "fc_port_info_t" },
1137 { "fc", "rport-login-end", 3, 3, "int", "int" },
1138 { "fc", "rport-login-end", 4, 4, "int", "int" },
1139 { "fc", "rport-logout-start", 0, 0, "fct_cmd_t",
1140   "conninfo_t" },
1141 { "fc", "rport-logout-start", 1, 1, "fct_local_port_t",
1142   "fc_port_info_t" },
1143 { "fc", "rport-logout-start", 2, 2, "fct_i_remote_port_t",
1144   "fc_port_info_t" },
1145 { "fc", "rport-logout-start", 3, 3, "int", "int" },
1146 { "fc", "rport-logout-end", 0, 0, "fct_cmd_t",
1147   "conninfo_t" },

```

```

1148 { "fc", "rport-logout-end", 1, 1, "fct_local_port_t **",
1149     "fc_port_info_t **" },
1150 { "fc", "rport-logout-end", 2, 2, "fct_i_remote_port_t **",
1151     "fc_port_info_t **" },
1152 { "fc", "rport-logout-end", 3, 3, "int", "int" },
1153 { "fc", "scsi-command", 0, 0, "fct_cmd_t **",
1154     "conninfo_t **" },
1155 { "fc", "scsi-command", 1, 1, "fct_i_local_port_t **",
1156     "fc_port_info_t **" },
1157 { "fc", "scsi-command", 2, 2, "scsi_task_t **",
1158     "scsicmd_t **" },
1159 { "fc", "scsi-command", 3, 3, "fct_i_remote_port_t **",
1160     "fc_port_info_t **" },
1161 { "fc", "scsi-response", 0, 0, "fct_cmd_t **",
1162     "conninfo_t **" },
1163 { "fc", "scsi-response", 1, 1, "fct_i_local_port_t **",
1164     "fc_port_info_t **" },
1165 { "fc", "scsi-response", 2, 2, "scsi_task_t **",
1166     "scsicmd_t **" },
1167 { "fc", "scsi-response", 3, 3, "fct_i_remote_port_t **",
1168     "fc_port_info_t **" },
1169 { "fc", "xfer-start", 0, 0, "fct_cmd_t **",
1170     "conninfo_t **" },
1171 { "fc", "xfer-start", 1, 1, "fct_i_local_port_t **",
1172     "fc_port_info_t **" },
1173 { "fc", "xfer-start", 2, 2, "scsi_task_t **",
1174     "scsicmd_t **" },
1175 { "fc", "xfer-start", 3, 3, "fct_i_remote_port_t **",
1176     "fc_port_info_t **" },
1177 { "fc", "xfer-start", 4, 4, "stmf_data_buf_t **",
1178     "fc_xferinfo_t **" },
1179 { "fc", "xfer-done", 0, 0, "fct_cmd_t **",
1180     "conninfo_t **" },
1181 { "fc", "xfer-done", 1, 1, "fct_i_local_port_t **",
1182     "fc_port_info_t **" },
1183 { "fc", "xfer-done", 2, 2, "scsi_task_t **",
1184     "scsicmd_t **" },
1185 { "fc", "xfer-done", 3, 3, "fct_i_remote_port_t **",
1186     "fc_port_info_t **" },
1187 { "fc", "xfer-done", 4, 4, "stmf_data_buf_t **",
1188     "fc_xferinfo_t **" },
1189 { "fc", "rscn-receive", 0, 0, "fct_i_local_port_t **",
1190     "conninfo_t **" },
1191 { "fc", "rscn-receive", 1, 1, "int", "int" },
1192 { "fc", "abts-receive", 0, 0, "fct_cmd_t **",
1193     "conninfo_t **" },
1194 { "fc", "abts-receive", 1, 1, "fct_i_local_port_t **",
1195     "fc_port_info_t **" },
1196 { "fc", "abts-receive", 2, 2, "fct_i_remote_port_t **",
1197     "fc_port_info_t **" },
1198
1199 { NULL }
1200 };
1201
1202 /*ARGSUSED*/
1203 void
1204 sdt_getargdesc(void *arg, dtrace_id_t id, void *parg, dtrace_argdesc_t *desc)
1205 {
1206     sdt_probe_t *sdp = parg;
1207     int i;
1208
1209     desc->dtargd_native[0] = '\0';
1210     desc->dtargd_xlate[0] = '\0';
1211
1212     for (i = 0; sdt_args[i].sda_provider != NULL; i++) {

```

```

1214     sdt_argdesc_t *a = &sdt_args[i];
1215
1216     if (strcmp(sdp->sdp_provider->sdt_name, a->sda_provider) != 0)
1217         continue;
1218
1219     if (a->sda_name != NULL &&
1220         strcmp(sdp->sdp_name, a->sda_name) != 0)
1221         continue;
1222
1223     if (desc->dtargd_ndx != a->sda_ndx)
1224         continue;
1225
1226     if (a->sda_native != NULL)
1227         (void) strcpy(desc->dtargd_native, a->sda_native);
1228
1229     if (a->sda_xlate != NULL)
1230         (void) strcpy(desc->dtargd_xlate, a->sda_xlate);
1231
1232     desc->dtargd_mapping = a->sda_mapping;
1233     return;
1234 }
1235
1236 desc->dtargd_ndx = DTRACE_ARGNONE;
1237 }

```

new/usr/src/uts/common/inet/Makefile

1

1853 Sat Aug 18 10:37:08 2012

new/usr/src/uts/common/inet/Makefile

dccp: finish moving headers

```
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 #
26 # uts/common/inet/Makefile
27 #
28 # include global definitions
29 include ../../../Makefile.master

31 HDRS=   arp.h common.h ipclassifier.h ip.h ip6.h ipdrop.h ipnet.h \
32         ipsec.h ipsecesp.h ipsec_info.h iptun.h ip6_asp.h ip_if.h ip_ire.h \
33         ip_multi.h ip_netinfo.h ip_ndp.h ip_rts.h ipsec_impl.h keysock.h \
34         led.h mi.h mib2.h nd.h optcom.h sadb.h sctp_itf.h snmpcom.h tcp.h \
35         tcp_sack.h tcp_stack.h tunables.h udp_impl.h rawip_impl.h ipp_common.h \
36         ip_ftable.h ip_impl.h ip_stack.h ip_arp.h tcp_impl.h wifi_ioctl.h \
37         ip2mac.h ip2mac_impl.h tcp_stats.h dccp.h dccp_impl.h dccp_ip.h \
38         dccp_stack.h dccp_stats.h
37         ip2mac.h ip2mac_impl.h tcp_stats.h

40 ROOTDIRS= $(ROOT)/usr/include/inet

42 ROOTHDRS= $(HDRS:%=$(ROOT)/usr/include/inet/%)

44 CHECKHDRS= $(HDRS:%.h=%.check)

46 $(ROOTDIRS)/%: %
47     $(INS.file)

49 .KEEP_STATE:

51 .PARALLEL: $(CHECKHDRS)

53 install_h: $(ROOTDIRS) $(ROOTHDRS)

55 $(ROOTDIRS):
56     $(INS.dir)

58 check:  $(CHECKHDRS)
```

new/usr/src/uts/common/inet/dccp.h

1

6239 Sat Aug 18 10:37:08 2012

new/usr/src/uts/common/inet/dccp.h

dccp: finish moving headers

```
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 */
11
12 /*
13  * Copyright 2012 David Hoeppner. All rights reserved.
14 */
15
16 #ifndef _INET_DCCP_H
17 #define _INET_DCCP_H
18
19 #include <sys/inttypes.h>
20 #include <sys/socket.h>
21 #include <sys/socket_proto.h>
22
23 #include <netinet/in.h>
24 #include <netinet/ip6.h>
25 #include <netinet/dccp.h>
26
27 #include <inet/common.h>
28 #include <inet/ip.h>
29 #include <inet/ip6.h>
30 #include <inet/tunables.h>
31 #include <inet/dccp_stack.h>
32 #include <inet/optcom.h>
33
34 #ifdef __cplusplus
35 extern "C" {
36 #endif
37
38 /*
39  * DCCP states
40 */
41 #define DCCPS_CLOSED 1
42 #define DCCPS_BOUND 2
43 #define DCCPS_REQUEST 3
44 #define DCCPS_LISTEN 4
45 #define DCCPS_PARTOPEN 5
46 #define DCCPS_RESPOND 6
47 #define DCCPS_OPEN 7
48 #define DCCPS_CLOSING 8
49 #define DCCPS_CLOSEREQ 9
50 #define DCCPS_TIMEWAIT 10
51
52 /*
53  * DCCP header structures.
54 */
55
56 /* Generic protocol header (RFC 4340, Section 5.1.) */
57 typedef struct dccphdr_s {
58     uint8_t dh_lport[2];
59     uint8_t dh_fport[2];
60     uint8_t dh_offset;
61     uint8_t dh_ccval;
62 }
```

new/usr/src/uts/common/inet/dccp.h

2

```
62     dh_cscov;
63     uint8_t db_sum[2];
64     uint8_t dh_reserved;
65     uint8_t dh_type;
66     uint8_t dh_x;
67     uint8_t dh_res_seq;
68     uint8_t dh_seq[2];
69 } dccph_t;
70
71 #define DCCP_HDR_LENGTH(dccph) \
72     (((dccph_t *)dccph)->dh_offset * 4) /* XXX >> 2 */
73 #define DCCP_MAX_HDR_LENGTH 1020
74 #define DCCP_MIN_HEADER_LENGTH 12
75
76 /* Generic protocol header aligned (RFC 4340, Section 5.1.) */
77 typedef struct dccphdra_s {
78     in_port_t dha_lport; /* Source port */
79     in_port_t dha_fport; /* Destination port */
80     uint8_t dha_offset; /* Data offset */
81     uint8_t dha_cscov; /* Checksum coverage */
82     uint8_t dha_ccval; /* */
83     uint16_t dha_sum; /* Checksum */
84     uint8_t dha_x; /* Reserved */
85     uint8_t dha_type; /* Packet type */
86     uint8_t dha_reserved; /* Header type */
87     uint8_t dha_res_seq;
88     uint16_t dha_seq; /* Partial sequence number */
89 } dccpha_t;
90
91 typedef struct dccphdra_ext_s {
92     uint32_t dha_ext_seq;
93 } dccpha_ext_t;
94
95 /* Acknowledgement number */
96 typedef struct dccphdra_ack {
97     uint16_t dha_ack_reserved;
98     uint16_t dha_ack_high;
99     uint32_t dha_ack_low;
100 } dccpha_ack_t;
101
102 /* Service number */
103 typedef struct dccphdra_srv {
104     uint32_t dha_srv_code;
105 } dccpha_srv_t;
106
107 /* Reset data */
108 typedef struct dccphdra_reset {
109     uint8_t dha_reset_code;
110     uint8_t dha_reset_data[3];
111 } dccpha_reset_t;
112
113 /*
114  * Control structure for each open TCP stream,
115  * defined only within the kernel or for a kmem user.
116  * NOTE: tcp_reinit_values MUST have a line for each field in this structure!
117  */
118 #if (defined(_KERNEL) || defined(_KMEMUSER))
119
120 /* Internal DCCP structure */
121 typedef struct dccp_s {
122
123     conn_t *dccp_connp; /* Backpointer to conn_t */
124     dccp_stack_t *dccp_dccps; /* Backpointer to dccp_stack_t */
125
126     int32_t dccp_state;
127 }
```

```

128     uint64_t      dccp_last_rcv_lbolt;

130     uint32_t      dccp_ibsegs;    /* Inbound segments on this stream */
131     uint32_t      dccp_obsegs;    /* Outbound segments on this stream */

133     uint32_t
134     dccp_hard_binding: 1,
135     dccp_loopback: 1,             /* Src and dst are the same machine */
136     dccp_localnet: 1,            /* Src and dst are on the same subnet */
137     dccp_active_open: 1,         /* This is a active open */
138     dccp_detached : 1,           /* If we're detached from a stream */
139     dccp_dummy: 1;

141     uint32_t
142     dccp_tconnind_started: 1,
143     dccp_dummy2: 1;

145     uint32_t
146     dccp_allow_short_seqnos: 1,
147     dccp_ecn_incapable: 1;

149     /*
150      * Timers and timestamps.
151      */
152     mblk_t      *dccp_timercache;    /* Timer cache */

154     timeout_id_t dccp_timer_tid;      /* Timer service id */
155     timeout_id_t dccp_ka_tid;         /* Keepalive timer id */

157     clock_t      dccp_timestamp_init; /* Time reference */
158     int32_t      dccp_timestamp_echo; /* Timestamp found in options */
159     clock_t      dccp_timestamp;

161     clock_t      dccp_first_timer_threshold;
162     clock_t      dccp_second_timer_threshold;
163     clock_t      dccp_first_ctimer_threshold;
164     clock_t      dccp_second_ctimer_threshold;

166     int32_t      dccp_ka_last_intrvl; /* Last probe interval */
167     uint32_t      dccp_ka_interval;    /* Keepalive interval */
168     uint32_t      dccp_ka_rinterval;   /* Keepalive retransmit */
169     uint32_t      dccp_ka_abort_thres; /* Abort threshold */
170     uint32_t      dccp_ka_cnt;         /* Probes counter */

172     int64_t      dccp_last_rcv_time;   /* Last segment rcv time */

174     int          dccp_conn_req_max;    /* # request allowed */

176     int32_t      dccp_client_errno;    /* How the client screwed up */

178     /*
179      * Bind related.
180      */
181     struct dccp_s *dccp_bind_hash;     /* Bind hash chain */
182     struct dccp_s *dccp_bind_hash_port; /* Bound to the same port */
183     struct dccp_s **dccp_ptpbhn;

185     struct dccphdra_s *dccp_dccpha;    /* Template header */

187     mblk_t      *dccp_xmit_head;

189     /*
190      * Pointers into the header template.
191      */
192     ipha_t      *dccp_ipha;
193     ip6_t      *dccp_ip6h;

```

```

195     t_uscalar_t  dccp_acceptor_id;    /* ACCEPTOR_id */

197     sock_connid_t dccp_connid;

199     /* Incrementing pending conn req ID */
200     t_scalar_t    dccp_conn_req_seqnum;

202     boolean_t     dccp_issocket;       /* This is a socket dccp */

204     /* List of features being negotiated */
205     list_t        dccp_features;

207     struct dccp_s *dccp_listener;      /* Our listener */
208     struct dccp_s *dccp_saved_listener; /* Saved listener */

210     kmutex_t      dccp_eager_lock;

212     /*
213      * Sequence numbers (Section 7.1.)
214      */
215     uint64_t      dccp_swl;            /* Sequence number window low */
216     uint64_t      dccp_swh;            /* Sequence number window high */
217     uint64_t      dccp_awl;            /* Ack number window low */
218     uint64_t      dccp_ahw;            /* Ack number window high */
219     uint64_t      dccp_iss;            /* Initial sequence number sent */
220     uint64_t      dccp_isr;            /* Initial sequence number received */
221     uint64_t      dccp_osr;            /* First OPEN sequence number */
222     uint64_t      dccp_gss;            /* Greatest sequence number sent */
223     uint64_t      dccp_gsr;            /* Greatest sequence */
224     /* number received */
225     uint64_t      dccp_gar;            /* Greatest acknowledgement */
226     /* number received */

228     uint64_t      dccp_sequence_window;

230     uint8_t      dccp_reset_code;
231     uint8_t      dccp_reset_data[3];
232 } dccp_t;

234 typedef struct dccp_df_s {
235     struct dccp_s *df_dccp;
236     kmutex_t      df_lock;
237     uchar_t       df_pad[TF_CACHEL_PAD - (sizeof (dccp_t *) +
238                                     sizeof (kmutex_t))];
239 } dccp_df_t;

241 #endif /* _KERNEL */

243 #ifdef __cplusplus
244 }
245 #endif

247 #endif /* _INET_DCCP_H */
248 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp.c

1

32398 Sat Aug 18 10:37:08 2012

new/usr/src/uts/common/inet/dccp/dccp.c

dccp: starting module template

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
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 Hoeppner. All rights reserved.
29 */
30
31 /*
32  * This file implements the Data Congestion Control Protocol (DCCP).
33 */
34
35 #include <sys/types.h>
36 #include <sys/stream.h>
37 #include <sys/stropts.h>
38 #include <sys/strlog.h>
39 #include <sys/strsun.h>
40 #define _SUN_TPI_VERSION 2
41 #include <sys/tihdr.h>
42 #include <sys/socket.h>
43 #include <sys/socketvar.h>
44 #include <sys/sockio.h>
45 #include <sys/priv.h>
46 #include <sys/vtrace.h>
47 #include <sys/sdt.h>
48 #include <sys/debug.h>
49 #include <sys/ddi.h>
50 #include <sys/isa_defs.h>
51 #include <sys/policy.h>
52 #include <sys/tsol/label.h>
53 #include <sys/tsol/tnet.h>
54
55 #include <inet/common.h>
56 #include <inet/ip.h>
57 #include <inet/ip_impl.h>
58 #include <inet/ip6.h>
59 #include <inet/dccp.h>
60 #include <inet/dccp_impl.h>
61 #include <inet/dccp_stack.h>
```

new/usr/src/uts/common/inet/dccp/dccp.c

2

```
62 #include <inet/kstatcom.h>
63 #include <inet/snmpcom.h>
64
65 #include <sys/cmn_err.h>
66
67 int dccp_squeue_flag;
68
69 /* Setable in /etc/system */
70 uint_t dccp_bind_fanout_size = DCCP_BIND_FANOUT_SIZE;
71
72 static void      dccp_notify(void *, ip_xmit_attr_t *, ixa_notify_type_t,
73                             ixa_notify_arg_t);
74
75 /* Functions to register netstack */
76 static void      *dccp_stack_init(netstackid_t, netstack_t *);
77 static void      dccp_stack_fini(netstackid_t, void *);
78
79 /* Stream device open functions */
80 static int      dccp_openv4(queue_t *, dev_t *, int, int, cred_t *);
81 static int      dccp_openv6(queue_t *, dev_t *, int, int, cred_t *);
82 static int      dccp_open(queue_t *, dev_t *, int, int, cred_t *,
83                           boolean_t);
84
85 /* Write service routine */
86 static void      dccp_wsrv(queue_t *);
87
88 /* Connection related functions */
89 static int      dccp_connect_ipv4(dccp_t *, ipaddr_t *, in_port_t, uint_t);
90 static int      dccp_connect_ipv6(dccp_t *, in6_addr_t *, in_port_t, uint32_t,
91                                   uint_t, uint32_t);
92
93 /* Initialise ISS */
94 static void      dccp_iss_init(dccp_t *);
95 static void      dccp_reinit(dccp_t *);
96
97 struct module_info dccp_rinfo = {
98     DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, DCCP_RECV_HIWATER,
99     DCCP_RECV_LOWATER
100 };
101
102 static struct module_info dccp_winfo = {
103     DCCP_MOD_ID, DCCP_MOD_NAME, 0, INFPSZ, 127, 16
104 };
105
106 /*
107  * Queue information structure with DCCP entry points.
108  */
109 struct qinit dccp_rinitv4 = {
110     NULL, (pfi_t)dccp_rsrv, dccp_openv4, dccp_tpi_close, NULL, &dccp_rinfo
111 };
112
113 struct qinit dccp_rinitv6 = {
114     NULL, (pfi_t)dccp_rsrv, dccp_openv6, dccp_tpi_close, NULL, &dccp_rinfo
115 };
116
117 struct qinit dccp_winit = {
118     (pfi_t)dccp_wput, (pfi_t)dccp_wsrv, NULL, NULL, NULL, &dccp_winfo
119 };
120
121 /* Initial entry point for TCP in socket mode */
122 struct qinit dccp_sock_winit = {
123     (pfi_t)dccp_wput_sock, (pfi_t)dccp_wsrv, NULL, NULL, NULL, &dccp_winfo
124 };
125
126 struct qinit dccp_fallback_sock_winit = {
127     (pfi_t)dccp_wput_fallback, NULL, NULL, NULL, NULL, &dccp_winfo
128 };
```

```

128 };
129 /*
130  * DCCP as acceptor STREAM.
131  */
132 struct qinit dccp_acceptor_rinit = {
133     NULL, (pfi_t)dccp_rsrv, NULL, dccp_tpi_close_accept, NULL, &dccp_winfo
134 };

136 struct qinit dccp_acceptor_winit = {
137     (pfi_t)dccp_tpi_accept, NULL, NULL, NULL, NULL, &dccp_winfo
138 };

140 /* AF_INET /dev/dccp */
141 struct streamtab dccpinfov4 = {
142     &dccp_rinitv4, &dccp_winit
143 };

145 /* AF_INET6 /dev/dccp6 */
146 struct streamtab dccpinfov6 = {
147     &dccp_rinitv6, &dccp_winit
148 };

150 /* Template for response to info request */
151 struct T_info_ack dccp_g_t_info_ack = {
152     T_INFO_ACK,          /* PRIM_type */
153     0,                   /* TSdu_size */
154     T_INFINITE,          /* ETSdu_size */
155     T_INVALID,           /* Cdata_size */
156     T_INVALID,           /* Ddata_size */
157     sizeof(sin_t),       /* ADDR_size */
158     0,                   /* OPT_size - not initialized here */
159     TIDUSZ,              /* TIDU_size */
160     T_COTS_ORD,          /* SERV_type */
161     DCCPS_CLOSED,        /* CURRENT_state */
162     (XPG4_1|EXPINLINE)   /* PROVIDER_flag */
163 };

165 struct T_info_ack dccp_g_t_info_ack_v6 = {
166     T_INFO_ACK,          /* PRIM_type */
167     0,                   /* TSdu_size */
168     T_INFINITE,          /* ETSdu_size */
169     T_INVALID,           /* Cdata_size */
170     T_INVALID,           /* Ddata_size */
171     sizeof(sin6_t),      /* ADDR_size */
172     0,                   /* OPT_size - not initialized here */
173     TIDUSZ,              /* TIDU_size */
174     T_COTS_ORD,          /* SERV_type */
175     DCCPS_CLOSED,        /* CURRENT_state */
176     (XPG4_1|EXPINLINE)   /* PROVIDER_flag */
177 };

179 /*
180  * DCCP Tunables.
181  */
182 extern mod_prop_info_t dccp_propinfo_tbl[];
183 extern int dccp_propinfo_count;

185 /*
186  * Register DCCP in ip netstack.
187  */
188 void
189 dccp_ddi_g_init(void)
190 {
191     /* Global timer cache */
192     dccp_timer_cache = kmem_cache_create("dccp_timer_cache",
193     sizeof(dccp_timer_t) + sizeof(mblk_t), 0,

```

```

194     NULL, NULL, NULL, NULL, NULL, 0);
195
196     netstack_register(NS_DCCP, dccp_stack_init, NULL, dccp_stack_fini);
197 }

199 /*
200  * Unregister DCCP from ip netstack.
201  */
202 void
203 dccp_ddi_g_destroy(void)
204 {
205     /* Global timer cache */
206     kmem_cache_destroy(dccp_timer_cache);

208     netstack_unregister(NS_DCCP);
209 }

211 #define INET_NAME        "ip"

213 /*
214  * Initialize this DCCP stack instance.
215  */
216 static void *
217 dccp_stack_init(netstackid_t stackid, netstack_t *ns)
218 {
219     dccp_stack_t *dccps;
220     major_t major;
221     size_t arrsz;
222     int error;
223     int i;

225     dccps = kmem_zalloc(sizeof(*dccps), KM_SLEEP);
226     if (dccps == NULL) {
227         return (NULL);
228     }
229     dccps->dccps_netstack = ns;

231     /* Ports */
232     mutex_init(&dccps->dccps_epriv_port_lock, NULL, MUTEX_DEFAULT, NULL);
233     dccps->dccps_num_epriv_ports = DCCP_NUM_EPRIV_PORTS;
234     dccps->dccps_epriv_ports[0] = ULP_DEF_EPRIV_PORT1;
235     dccps->dccps_epriv_ports[1] = ULP_DEF_EPRIV_PORT2;
236     dccps->dccps_min_anonpriv_port = 512;

238     dccps->dccps_bind_fanout_size = dccp_bind_fanout_size;

240     /* Bind fanout */
241     dccps->dccps_bind_fanout = kmem_zalloc(dccps->dccps_bind_fanout_size *
242     sizeof(dccp_df_t), KM_SLEEP);
243     for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
244         mutex_init(&dccps->dccps_bind_fanout[i].df_lock, NULL,
245         MUTEX_DEFAULT, NULL);
246     }

248     /* Tunable properties */
249     arrsz = dccp_propinfo_count * sizeof(mod_prop_info_t);
250     dccps->dccps_propinfo_tbl = kmem_alloc(arrsz, KM_SLEEP);
251     if (dccps->dccps_propinfo_tbl == NULL) {
252         kmem_free(dccps, sizeof(*dccps));
253         return (NULL);
254     }
255     bcopy(dccp_propinfo_tbl, dccps->dccps_propinfo_tbl, arrsz);

257     /* Allocate per netstack cpu stats */
258     mutex_enter(&cpu_lock);
259     dccps->dccps_sc_cnt = MAX(ncpus, boot_ncpus);

```

```

260     mutex_exit(&cpu_lock);

262     dccps->dccps_sc = kmem_zalloc(max_ncpus * sizeof (dccp_stats_cpu_t *),
263     KM_SLEEP);
264     for (i = 0; i < dccps->dccps_sc_cnt; i++) {
265         dccps->dccps_sc[i] = kmem_zalloc(sizeof (dccp_stats_cpu_t),
266         KM_SLEEP);
267     }

269     /* Kernel statistics */
270     //dccps->dccps_kstat = dccp_kstat2_init(stackid);
271     //dccps->dccps_mibkp = dccp_kstat_init(stackid);

273     /* Driver major number */
274     major = mod_name_to_major(INET_NAME);
275     error = ldi_ident_from_major(major, &dccps->dccps_ldi_ident);
276     ASSERT(error == 0);

278     return (dccps);
279 }

281 /*
282  * Destroy this DCCP netstack instance.
283  */
284 static void
285 dccp_stack_fini(netstackid_t stackid, void *arg)
286 {
287     dccp_stack_t    *dccps = (dccp_stack_t *)arg;
288     int              i;

290     /* Free cpu stats */
291     for (i = 0; i < dccps->dccps_sc_cnt; i++) {
292         kmem_free(dccps->dccps_sc[i], sizeof (dccp_stats_cpu_t));
293     }
294     kmem_free(dccps->dccps_sc, max_ncpus * sizeof (dccp_stats_cpu_t *));

296     /* Free tunable properties */
297     kmem_free(dccps->dccps_propinfo_tbl,
298     dccp_propinfo_count * sizeof (mod_prop_info_t));
299     dccps->dccps_propinfo_tbl = NULL;

301     /* Free bind fanout */
302     for (i = 0; i < dccps->dccps_bind_fanout_size; i++) {
303         ASSERT(dccps->dccps_bind_fanout[i].df_dccp == NULL);
304         mutex_destroy(&dccps->dccps_bind_fanout[i].df_lock);
305     }
306     kmem_free(dccps->dccps_bind_fanout, dccps->dccps_bind_fanout_size *
307     sizeof (dccp_df_t));
308     dccps->dccps_bind_fanout = NULL;

310     /* Kernel statistics */
311     dccp_kstat_fini(stackid, dccps->dccps_mibkp);
312     dccps->dccps_mibkp = NULL;
313     dccp_kstat2_fini(stackid, dccps->dccps_kstat);
314     dccps->dccps_kstat = NULL;

316     ldi_ident_release(dccps->dccps_ldi_ident);

318     kmem_free(dccps, sizeof (*dccps));
319 }

321 /* /dev/dccp */
322 static int
323 dccp_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
324 {
325     cmn_err(CE_NOTE, "dccp.c: dccp_openv4\n");

```

```

327     return (dccp_open(q, devp, flag, sflag, credp, B_FALSE));
328 }

330 /* /dev/dccp6 */
331 static int
332 dccp_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
333 {
334     cmn_err(CE_NOTE, "dccp.c: dccp_openv6\n");

336     return (dccp_open(q, devp, flag, sflag, credp, B_TRUE));
337 }

339 /*
340  * Common open function for v4 and v6 devices.
341  */
342 static int
343 dccp_open(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp,
344     boolean_t isv6)
345 {
346     conn_t          *connp;
347     dccp_t           *dccp;
348     vmem_t           *minor_arena;
349     dev_t            conn_dev;
350     boolean_t        issocket;
351     int              error;

353     cmn_err(CE_NOTE, "dccp.c: dccp_open");

355     /* If the stream is already open, return immediately */
356     if (q->q_ptr != NULL) {
357         return (0);
358     }

360     if (sflag == MODOPEN) {
361         return (EINVAL);
362     }

364     if ((ip_minor_arena_la != NULL) && (flag & SO_SOCKSTR) &&
365     ((conn_dev = inet_minor_alloc(ip_minor_arena_la)) != 0)) {
366         minor_arena = ip_minor_arena_la;
367     } else {
368         /*
369          * Either minor numbers in the large arena were exhausted
370          * or a non socket application is doing the open.
371          * Try to allocate from the small arena.
372          */
373         if ((conn_dev = inet_minor_alloc(ip_minor_arena_sa)) == 0) {
374             return (EBUSY);
375         }
376         minor_arena = ip_minor_arena_sa;
377     }

379     ASSERT(minor_arena != NULL);

381     *devp = makedevice(getmajor(*devp), (minor_t)conn_dev);

383     if (flag & SO_FALLBACK) {
384         /*
385          * Non streams socket needs a stream to fallback to.
386          */
387         RD(q)->q_ptr = (void *)conn_dev;
388         WR(q)->q_qinfo = &dccp_fallback_sock_winit;
389         WR(q)->q_ptr = (void *)minor_arena;
390         qprocson(q);
391         return (0);

```



```

392     } else if (flag & SO_ACCEPTOR) {
393         q->q_qinfo = &dccp_acceptor_rinit;
394         /*
395          * The conn_dev and minor_arena will be subsequently used by
396          * dccp_tli_accept() and dccp_tpi_close_accept() to figure out
397          * the minor device number for this connection from the q_ptr.
398          */
399         RD(q)->q_ptr = (void *)conn_dev;
400         WR(q)->q_qinfo = &dccp_acceptor_winit;
401         WR(q)->q_ptr = (void *)minor_arena;
402         qprocson(q);
403         return (0);
404     }
406     isssocket = flag & SO_SOCKSTR;
407     connp = dccp_create_common(credp, isv6, isssocket, &error);
408     if (connp == NULL) {
409         inet_minor_free(minor_arena, conn_dev);
410         q->q_ptr = WR(q)->q_ptr = NULL;
411         return (error);
412     }
414     connp->conn_rq = q;
415     connp->conn_wq = WR(q);
416     q->q_ptr = WR(q)->q_ptr = connp;
418     connp->conn_dev = conn_dev;
419     connp->conn_minor_arena = minor_arena;
421     ASSERT(q->q_qinfo == &dccp_rinitv4 || q->q_qinfo == &dccp_rinitv6);
422     ASSERT(WR(q)->q_qinfo == &dccp_winit);
424     dccp = connp->conn_dccp;
426     if (isssocket) {
427         WR(q)->q_qinfo = &dccp_sock_winit;
428     } else {
429 #ifdef _ILP32
430         dccp->dccp_acceptor_id = (t_uscalar_t)RD(q);
431 #else
432         dccp->dccp_acceptor_id = conn_dev;
433 #endif /* _ILP32 */
434     }
436     /*
437     * Put the ref for DCCP. Ref for IP was already put
438     * by ipcl_conn_create. Also Make the conn_t globally
439     * visible to walkers.
440     */
441     mutex_enter(&connp->conn_lock);
442     CONN_INC_REF_LOCKED(connp);
443     ASSERT(connp->conn_ref == 2);
444     connp->conn_state_flags &= -CONN_INCIPIENT;
445     mutex_exit(&connp->conn_lock);
447     qprocson(q);
449     return (0);
450 }
452 /*
453  * IXA notify
454  */
455 static void
456 dccp_notify(void *arg, ip_xmit_attr_t *ixa, ix_notify_type_t ntype,
457             ix_notify_arg_t narg)

```

```

458 {
459     cmn_err(CE_NOTE, "dccp.c: dccp_notify");
460 }
462 /*
463  * Build the template headers.
464  */
465 int
466 dccp_build_hdrs(dccp_t *dccp)
467 {
468     dccp_stack_t    *dccps = dccp->dccp_dccps;
469     conn_t           *connp = dccp->dccp_connp;
470     dccpha_t         *dccpha;
471     uint32_t         cksun;
472     char             buf[DCCP_MAX_HDR_LENGTH];
473     uint_t            buflen;
474     uint_t            ulplen = 12;
475     uint_t            extralen = 0;
476     int               error;
478     cmn_err(CE_NOTE, "dccp.c: dccp_build_hdrs");
480     buflen = connp->conn_ht_ulp_len;
481     if (buflen != 0) {
482         cmn_err(CE_NOTE, "buflen != 0");
483         bcopy(connp->conn_ht_ulp, buf, buflen);
484         extralen -= buflen - ulplen;
485         ulplen = buflen;
486     }
488     mutex_enter(&connp->conn_lock);
489     error = conn_build_hdr_template(connp, ulplen, extralen,
490                                     &connp->conn_laddr_v6, &connp->conn_faddr_v6, connp->conn_flowinfo);
491     mutex_exit(&connp->conn_lock);
492     if (error != 0) {
493         cmn_err(CE_NOTE, "conn_build_hdr_template failed");
494         return (error);
495     }
497     dccpha = (dccpha_t *)connp->conn_ht_ulp;
498     dccp->dccp_dccpha = dccpha;
500     if (buflen != 0) {
501         bcopy(buf, connp->conn_ht_ulp, buflen);
502     } else {
503         dccpha->dha_sum = 0;
504         dccpha->dha_lport = connp->conn_lport;
505         dccpha->dha_fport = connp->conn_fport;
506     }
508     cksun = sizeof(dccpha_t) + connp->conn_sum;
509     cksun = (cksun >> 16) + (cksun & 0xFFFF);
510     dccpha->dha_sum = htons(cksun);
511     dccpha->dha_offset = 7;
512     dccpha->dha_x = 1;
514     if (connp->conn_ipversion == IPV4_VERSION) {
515         dccp->dccp_ipha = (ipha_t *)connp->conn_ht_iphc;
516     } else {
517         dccp->dccp_ip6h = (ip6_t *)connp->conn_ht_iphc;
518     }
520     /* XXX */
522     return (0);
523 }

```

```

525 /*
526  * DCCP write service routine.
527  */
528 static void
529 dccp_wsrv(queue_t *q)
530 {
531     dccp_stack_t *dccps = Q_TO_DCCP(q)->dccp_dccps;
532
533     DCCP_STAT(dccps, dccp_wsrv_called);
534 }
535
536 /*
537  * Common create function for streams and sockets.
538  */
539 conn_t *
540 dccp_create_common(cred_t *credp, boolean_t isv6, boolean_t issocket,
541     int *errorp)
542 {
543     conn_t *connp;
544     dccp_t *dccp;
545     dccp_stack_t *dccps;
546     netstack_t *ns;
547     squeue_t *sqp;
548     zoneid_t zoneid;
549     int error;
550
551     cmn_err(CE_NOTE, "dccp.c: dccp_create_common\n");
552
553     ASSERT(errorp != NULL);
554
555     error = secpolicy_basic_net_access(credp);
556     if (error != 0) {
557         *errorp = error;
558         return (NULL);
559     }
560
561     /*
562      * Find the right netstack.
563      */
564     ns = netstack_find_by_cred(credp);
565     ASSERT(ns != NULL);
566     dccps = ns->netstack_dccp;
567     ASSERT(dccps != NULL);
568
569     /*
570      * For exclusive stacks we set the zoneid to zero
571      * to make TCP operate as if in the global zone.
572      */
573     if (ns->netstack_stackid != GLOBAL_NETSTACKID) {
574         zoneid = GLOBAL_ZONEID;
575     } else {
576         zoneid = crgetzoneid(credp);
577     }
578
579     sqp = IP_SQUEUE_GET((uint_t)gethrtime());
580     connp = (conn_t *)dccp_get_conn(sqp, dccps);
581     netstack_rele(dccps->dccps_netstack);
582     if (connp == NULL) {
583         *errorp = ENOSR;
584         return (NULL);
585     }
586     ASSERT(connp->conn_ixa->ixa_protocol == connp->conn_proto);
587
588     connp->conn_sqp = sqp;
589     connp->conn_initial_sqp = connp->conn_sqp;

```

```

590     connp->conn_ixa->ixa_sqp = connp->conn_sqp;
591     dccp = connp->conn_dccp;
592
593     /* Setting flags for ip output */
594     connp->conn_ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
595         IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO;
596
597     ASSERT(connp->conn_proto == IPPROTO_DCCP);
598     ASSERT(connp->conn_dccp == dccp);
599     ASSERT(dccp->dccp_connp == connp);
600
601     if (isv6) {
602         connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
603         connp->conn_ipversion = IPV6_VERSION;
604         connp->conn_family = AF_INET6;
605         /* XXX mms, ttl */
606     } else {
607         connp->conn_ipversion = IPV4_VERSION;
608         connp->conn_family = AF_INET;
609         /* XXX mms, ttl */
610     }
611     connp->conn_xmit_ipp.ipp_unicast_hops = connp->conn_default_ttl;
612
613     crhold(credp);
614     connp->conn_cred = credp;
615     connp->conn_cpuid = curproc->p_pid;
616     connp->conn_open_time = ddi_get_lbolt64();
617
618     ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
619     connp->conn_ixa->ixa_cred = credp;
620     connp->conn_ixa->ixa_cpuid = connp->conn_cpuid;
621
622     connp->conn_zoneid = zoneid;
623     connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);
624     connp->conn_ixa->ixa_zoneid = zoneid;
625     connp->conn_mlp_type = mlptSingle;
626
627     dccp->dccp_dccps = dccps;
628     dccp->dccp_state = DCCPS_CLOSED;
629
630     ASSERT(connp->conn_netstack == dccps->dccps_netstack);
631     ASSERT(dccp->dccp_dccps == dccps);
632
633     /*
634      * If the caller has the process-wide flag set, then default to MAC
635      * exempt mode. This allows read-down to unlabeled hosts.
636      */
637     if (getpflags(NET_MAC_AWARE, credp) != 0) {
638         connp->conn_mac_mode = CONN_MAC_AWARE;
639     }
640
641     if (issocket) {
642         dccp->dccp_issocket = 1;
643     }
644
645     /* XXX rcvbuf, sndbuf etc */
646
647     connp->conn_so_type = SOCK_STREAM;
648
649     SOCK_CONNID_INIT(dccp->dccp_connid);
650     dccp_init_values(dccp, NULL);
651
652     return (connp);
653 }
654
655 /*

```

```

656 * Common close function for streams and sockets.
657 */
658 void
659 dccp_close_common(conn_t *connp, int flags)
660 {
661     dccp_t      *dccp = connp->conn_dccp;
662     mblk_t      *mp;
663     boolean_t    conn_ioctl_cleanup_reqd = B_FALSE;
664
665     cmn_err(CE_NOTE, "dccp.c: dccp_close_common");
666
667     ASSERT(connp->conn_ref >= 2);
668
669     /*
670      * Mark the conn as closing. ipsq_pending_mp_add will not
671      * add any mp to the pending mp list, after this conn has
672      * started closing.
673      */
674     mutex_enter(&connp->conn_lock);
675     connp->conn_state_flags |= CONN_CLOSING;
676
677     if (connp->conn_oper_pending_ill != NULL) {
678         conn_ioctl_cleanup_reqd = B_TRUE;
679     }
680
681     CONN_INC_REF_LOCKED(connp);
682     mutex_exit(&connp->conn_lock);
683
684     ASSERT(connp->conn_ref >= 3);
685
686     /*
687      * Cleanup any queued ioctls here. This must be done before the wq/rq
688      * are re-written by dccp_close_output().
689      */
690     if (conn_ioctl_cleanup_reqd) {
691         conn_ioctl_cleanup(connp);
692     }
693
694     mutex_enter(&connp->conn_lock);
695     while (connp->conn_ioctlref > 0) {
696         cv_wait(&connp->conn_cv, &connp->conn_lock);
697     }
698     ASSERT(connp->conn_ioctlref == 0);
699     ASSERT(connp->conn_oper_pending_ill == NULL);
700     mutex_exit(&connp->conn_lock);
701
702     /* generate close */
703     /*
704      * SQUEUE_ENTER_ONE(connp->conn_sq, mp, dccp_close_output, connp,
705      * NULL, dccp_queue_flag, SQTAG_IP_DCCP_CLOSE);
706      */
707     /*
708      * nowait:
709      * connp->conn_cpid = NOPID;
710      */
711 }
712
713 /*
714 * Common bind function.
715 */
716 int
717 dccp_do_bind(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
718             boolean_t bind_to_req_port_only)
719 {
720     dccp_t      *dccp = connp->conn_dccp;
721     int          error;

```

```

723     cmn_err(CE_NOTE, "dccp.c: dccp_do_bind");
724
725     if (dccp->dccp_state >= DCCPS_BOUND) {
726         if (connp->conn_debug) {
727             (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
728                 "dccp_bind: bad state, %d", dccp->dccp_state);
729         }
730         return (-TOUTSTATE);
731     }
732
733     error = dccp_bind_check(connp, sa, len, cr, bind_to_req_port_only);
734     if (error != 0) {
735         return (error);
736     }
737
738     ASSERT(dccp->dccp_state == DCCPS_LISTEN);
739     /* XXX dccp_conn_req_max = 0 */
740
741     return (0);
742 }
743
744 /*
745 * Common unbind function.
746 */
747 int
748 dccp_do_unbind(conn_t *connp)
749 {
750     dccp_t      *dccp = connp->conn_dccp;
751     int32_t      oldstate;
752
753     cmn_err(CE_NOTE, "dccp.c: dccp_do_unbind");
754
755     switch (dccp->dccp_state) {
756     case DCCPS_OPEN:
757     case DCCPS_LISTEN:
758         break;
759     default:
760         return (-TOUTSTATE);
761     }
762
763     connp->conn_laddr_v6 = ipv6_all_zeros;
764     connp->conn_saddr_v6 = ipv6_all_zeros;
765
766     dccp_bind_hash_remove(dccp);
767
768     oldstate = dccp->dccp_state;
769     dccp->dccp_state = DCCPS_CLOSED;
770     DTRACE_DCCP6(state_change, void, NULL, ip_xmit_attr_t *,
771         connp->conn_ixa, void, NULL, dccp_t *, dccp, void, NULL,
772         int32_t, oldstate);
773
774     ip_unbind(connp);
775     bzero(&connp->conn_ports, sizeof (connp->conn_ports));
776
777     return (0);
778 }
779
780 /*
781 * Common listen function.
782 */
783 int
784 dccp_do_listen(conn_t *connp, struct sockaddr *sa, socklen_t len,
785               int backlog, cred_t *cr, boolean_t bind_to_req_port_only)
786 {
787     dccp_t      *dccp = connp->conn_dccp;

```

```

788 dccp_stack_t *dccps = dccp->dccp_dccps;
789 int32_t oldstate;
790 int error;

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

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

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

799     if ((dccp->dccp_state == DCCPS_BOUND ||
800         dccp->dccp_state == DCCPS_LISTEN) && backlog > 0) {
801         goto do_listen;
802     }
803     cmn_err(CE_NOTE, "DCCPS_BOUND, bad state");

805     if (connp->conn_debug) {
806         (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
807             "dccp_listen: bad state, %d", dccp->dccp_state);
808     }
809     return (-TOUTSTATE);
810 } else {
811     if (sa == NULL) {
812         sin6_t addr;
813         sin6_t *sin6;
814         sin_t *sin;

816         ASSERT(IPCL_IS_NONSTR(connp));

818         if (connp->conn_family == AF_INET) {
819             len = sizeof(sin_t);
820             sin = (sin_t *)&addr;
821             *sin = sin_null;
822             sin->sin_family = AF_INET;
823         } else {
824             ASSERT(connp->conn_family == AF_INET6);

826             len = sizeof(sin6_t);
827             sin6 = (sin6_t *)&addr;
828             *sin6 = sin6_null;
829             sin6->sin6_family = AF_INET6;
830         }

832         sa = (struct sockaddr *)&addr;
833     }

835     error = dccp_bind_check(connp, sa, len, cr,
836         bind_to_req_port_only);
837     if (error != 0) {
838         cmn_err(CE_NOTE, "dccp_bind_check failed");
839         return (error);
840     }
841     /* Fall through and do the fanout insertion */
842 }

844 do_listen:
845     ASSERT(dccp->dccp_state == DCCPS_BOUND ||
846         dccp->dccp_state == DCCPS_LISTEN);

848     dccp->dccp_conn_req_max = backlog;
849     if (dccp->dccp_conn_req_max) {
850         if (dccp->dccp_state != DCCPS_LISTEN) {
851             dccp->dccp_state = DCCPS_LISTEN;
852             DTRACE_DCCP6(state_change, void, NULL,
853                 ip_xmit_attr_t *, connp->conn_ixa, void, NULL,

```

```

854         dccp_t *, dccp, void, NULL, int32_t, DCCPS_BOUND);

856         dccp->dccp_second_ctimer_threshold =
857             dccps->dccps_ip_abort_linterval;
858     }
859 }

861 /* XXX */

863 connp->conn_rcv = dccp_input_listener_unbound;

865 /* Insert into the classifier table */
866 error = ip_laddr_fanout_insert(connp);
867 if (error != 0) {
868     /* Error - undo the bind */
869     oldstate = dccp->dccp_state;
870     dccp->dccp_state = DCCPS_CLOSED;
871     DTRACE_DCCP6(state_change, void, NULL, ip_xmit_attr_t *,
872         connp->conn_ixa, void, NULL, dccp_t *, dccp, void, NULL,
873         int32_t, oldstate);

875     connp->conn_bound_addr_v6 = ipv6_all_zeros;
876     connp->conn_laddr_v6 = ipv6_all_zeros;
877     connp->conn_saddr_v6 = ipv6_all_zeros;
878     connp->conn_ports = 0;

880     if (connp->conn_anon_port) {
881         zone_t *zone;

883         zone = crgetzone(cr);
884         connp->conn_anon_port = B_FALSE;
885         (void) tsol_mlp_anon(zone, connp->conn_mlp_type,
886             connp->conn_proto, connp->conn_lport, B_FALSE);
887     }
888     connp->conn_mlp_type = mlptSingle;

890     dccp_bind_hash_remove(dccp);

892     return (error);
893 } else {
894     /* XXX connection limits */
895 }

897     return (error);
898 }

900 /*
901  * Common connect function.
902  */
903 int
904 dccp_do_connect(conn_t *connp, const struct sockaddr *sa, socklen_t len,
905     cred_t *cr, pid_t pid)
906 {
907     dccp_t *dccp = connp->conn_dccp;
908     dccp_stack_t *dccps = dccp->dccp_dccps;
909     ip_xmit_attr_t *ixa = connp->conn_ixa;
910     mblk_t *req_mp;
911     sin_t *sin = (sin_t *)sa;
912     sin6_t *sin6 = (sin6_t *)sa;
913     ipaddr_t *dstaddrp;
914     in_port_t dstport;
915     uint_t srcid;
916     int32_t oldstate;
917     int error;

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

```

```

921     oldstate = dccp->dccp_state;

923     switch (len) {
924     case sizeof (sin_t):
925         sin = (sin_t *)sa;
926         if (sin->sin_port == 0) {
927             return (-TBADADDR);
928         }
929         if (connp->conn_ip6_v6only) {
930             return (EAFNOSUPPORT);
931         }
932         break;

934     case sizeof (sin6_t):
935         sin6 = (sin6_t *)sa;
936         if (sin6->sin6_port == 0) {
937             return (-TBADADDR);
938         }
939         break;

941     default:
942         return (EINVAL);
943     }

945     if (connp->conn_family == AF_INET6 &&
946         connp->conn_ipversion == IPV6_VERSION &&
947         IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
948         if (connp->conn_ip6_v6only) {
949             return (EADDRNOTAVAIL);
950         }

952         connp->conn_ipversion = IPV4_VERSION;
953     }

955     switch (dccp->dccp_state) {
956     case DCCPS_LISTEN:
957         /*
958          * Listening sockets are not allowed to issue connect().
959          */
960         if (IPCL_IS_NONSTR(connp)) {
961             return (EOPNOTSUPP);
962         }

964     case DCCPS_CLOSED:
965         /*
966          * We support quick connect.
967          */
968         /* FALLTHRU */
969     case DCCPS_OPEN:
970         break;

972     default:
973         return (-TOUTSTATE);
974     }

976     /*
977     * We update our cred/cpid based on the caller of connect.
978     */
979     if (connp->conn_cred != cr) {
980         crhold(cr);
981         crfree(connp->conn_cred);
982         connp->conn_cred = cr;
983     }
984     connp->conn_cpid = pid;

```

```

986     /* Cache things in the ixa without any refhold */
987     ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
988     ixa->ixa_cred = cr;
989     ixa->ixa_cpid = pid;

991     if (is_system_labeled()) {
992         ip_xmit_attr_restore_tsl(ixa, ixa->ixa_cred);
993     }

995     if (connp->conn_family == AF_INET6) {
996         if (!IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
997             error = dccp_connect_ipv6(dccp, &sin6->sin6_addr,
998                 sin6->sin6_port, sin6->sin6_flowinfo,
999                 sin6->__sin6_src_id, sin6->sin6_scope_id);
1000         } else {
1001             /*
1002              * Destination address is mapped IPv6 address.
1003              * Source bound address should be unspecified or
1004              * IPv6 mapped address as well.
1005              */
1006             if (!IN6_IS_ADDR_UNSPECIFIED(
1007                 &connp->conn_bound_addr_v6) &&
1008                 !IN6_IS_ADDR_V4MAPPED(&connp->conn_bound_addr_v6)) {
1009                 return (EADDRNOTAVAIL);
1010             }

1012             dstaddrp = &V4_PART_OF_V6((sin6->sin6_addr));
1013             dstport = sin6->sin6_port;
1014             srcid = sin6->__sin6_src_id;
1015             error = dccp_connect_ipv4(dccp, dstaddrp, dstport,
1016                 srcid);
1017         }
1018     } else {
1019         dstaddrp = &sin->sin_addr.s_addr;
1020         dstport = sin->sin_port;
1021         srcid = 0;
1022         error = dccp_connect_ipv4(dccp, dstaddrp, dstport, srcid);
1023     }

1025     if (error != 0) {
1026         cmn_err(CE_NOTE, "dccp_connect_ip failed");
1027         goto connect_failed;
1028     }

1030     /* XXX cluster */

1032     /* Connect succeeded */
1033     DCCPS_BUMP_MIB(dccps, dccpActiveOpens);
1034     dccp->dccp_active_open = 1;

1036     DTRACE_DCCP6(state_change, void, NULL, ip_xmit_attr_t *,
1037         connp->conn_ixa, void, NULL, dccp_t *, dccp, void, NULL,
1038         int32_t, DCCPS_BOUND);

1040     DCCP_TIMER_RESTART(dccp, 100);
1041     req_mp = dccp_generate_request(connp);
1042     if (req_mp != NULL) {
1043         /*
1044          * We must bump the generation before sending the request
1045          * to ensure that we use the right generation in case
1046          * this thread issues a "connected" up call.
1047          */
1048         SOCK_CONNID_BUMP(dccp->dccp_connid);

1050         DTRACE_DCCP5(connect_request, mblk_t *, NULL,
1051             ip_xmit_attr_t *, connp->conn_ixa,

```

```

1052         void_ip_t *, req_mp->b_rptr, dccp_t *, dccp,
1053         dccpha_t *,
1054         &req_mp->b_rptr[connp->conn_ix->ixa_ip_hdr_length]);

1056         dccp_send_data(dccp, req_mp);
1057     }

1059     return (0);

1061 connect_failed:
1062     cmn_err(CE_NOTE, "dccp_do_connect failed");

1064     connp->conn_faddr_v6 = ipv6_all_zeros;
1065     connp->conn_fport = 0;
1066     dccp->dccp_state = oldstate;

1068     /* XXX */
1069     return (error);
1070 }

1072 /*
1073  * Init values of a connection.
1074  */
1075 void
1076 dccp_init_values(dccp_t *dccp, dccp_t *parent)
1077 {
1078     conn_t      *connp = dccp->dccp_connp;
1079     dccp_stack_t *dccps = dccp->dccp_dccps;

1081     ASSERT((connp->conn_family == AF_INET &&
1082             connp->conn_ipversion == IPV4_VERSION) ||
1083            (connp->conn_family == AF_INET6 &&
1084             connp->conn_ipversion == IPV4_VERSION ||
1085             connp->conn_ipversion == IPV6_VERSION));

1087     if (parent == NULL) {

1089         dccp->dccp_first_ctimer_threshold =
1090             dccps->dccps_ip_notify_cinterval;
1091         dccp->dccp_second_ctimer_threshold =
1092             dccps->dccps_ip_abort_cinterval;
1093         dccp->dccp_first_timer_threshold =
1094             dccps->dccps_ip_notify_interval;
1095         dccp->dccp_second_timer_threshold =
1096             dccps->dccps_ip_abort_interval;

1098         dccp->dccp_ka_interval =
1099             dccps->dccps_keepalive_interval;
1100         dccp->dccp_ka_abort_thres =
1101             dccps->dccps_keepalive_abort_interval;
1102         dccp->dccp_ka_cnt = 0;
1103         dccp->dccp_ka_rinterval = 0;
1104     } else {
1105         /* Inherit various DCCP parameters from the parent */
1106         dccp->dccp_first_ctimer_threshold =
1107             parent->dccp_first_ctimer_threshold;
1108         dccp->dccp_second_ctimer_threshold =
1109             parent->dccp_second_ctimer_threshold;
1110         dccp->dccp_first_timer_threshold =
1111             parent->dccp_first_timer_threshold;
1112         dccp->dccp_second_timer_threshold =
1113             parent->dccp_second_timer_threshold;

1115         dccp->dccp_ka_interval = parent->dccp_ka_interval;
1116         dccp->dccp_ka_abort_thres = parent->dccp_ka_abort_thres;
1117         dccp->dccp_ka_cnt = parent->dccp_ka_cnt;

```

```

1118         dccp->dccp_ka_rinterval = parent->dccp_ka_rinterval;
1119     }

1121     dccp->dccp_last_rcv_time = ddi_get_lbolt();
1122     dccp->dccp_sequence_window = 100;

1124     connp->conn_mlp_type = mlptSingle;

1126     if (!connp->conn_debug) {
1127         connp->conn_debug = dccps->dccps_dbg;
1128     }
1129 }

1131 /*
1132  * Free dccp structure.
1133  */
1134 void
1135 dccp_free(dccp_t *dccp)
1136 {
1137     conn_t      *connp = dccp->dccp_connp;

1139     cmn_err(CE_NOTE, "dccp.c: dccp_free");

1141     connp->conn_rq = NULL;
1142     connp->conn_wq = NULL;

1144     if (connp->conn_upper_handle != NULL) {
1145         if (IPCL_IS_NONSTR(connp)) {
1146             (*connp->conn_upcalls->su_closed)(
1147                 connp->conn_upper_handle);
1148             dccp->dccp_detached = B_TRUE;
1149         }

1151         connp->conn_upper_handle = NULL;
1152         connp->conn_upcalls = NULL;
1153     }
1154 }

1156 void *
1157 dccp_get_conn(void *arg, dccp_stack_t *dccps)
1158 {
1159     conn_t      *connp;
1160     dccp_t      *dccp = NULL;
1161     queue_t      *sqp = (queue_t *)arg;
1162     netstack_t   *ns;

1164     /* XXX timewait */

1166     connp = ipcl_conn_create(IPCL_DCCPCONN, KM_NOSLEEP,
1167                             dccps->dccps_netstack);
1168     if (connp == NULL) {
1169         return (NULL);
1170     }

1172     dccp = connp->conn_dccp;
1173     dccp->dccp_dccps = dccps;

1175     /* List of features being negotiated */
1176     list_create(&dccp->dccp_features, sizeof (dccp_feature_t),
1177               offsetof(dccp_feature_t, df_next));

1179     connp->conn_rcv = dccp_input_data;
1180     connp->conn_rcvicmp = dccp_icmp_input;
1181     connp->conn_verifyicmp = dccp_verifyicmp;

1183     connp->conn_ix->ixa_notify = dccp_notify;

```

```

1184     connp->conn_ixa->ixa_notify_cookie = dccp;
1186     return ((void *)connp);
1187 }
1189 /*
1190  * Collect protocol properties to send to the upper handle.
1191  */
1192 void
1193 dccp_get_proto_props(dccp_t *dccp, struct sock_proto_props *sopp)
1194 {
1195     conn_t *connp = dccp->dccp_connp;
1197     cmn_err(CE_NOTE, "dccp.c: dccp_get_proto_props");
1199     sopp->sopp_flags = SOCKOPT_RCVHIWAT | SOCKOPT_MAXBLK | SOCKOPT_WROFF;
1200 }
1202 /*
1203  * IPv4 connect.
1204  */
1205 static int
1206 dccp_connect_ipv4(dccp_t *dccp, ipaddr_t *dstaddrp, in_port_t dstport,
1207                  uint_t srcid)
1208 {
1209     conn_t *connp = dccp->dccp_connp;
1210     dccp_stack_t *dccps = dccp->dccp_dccps;
1211     ipaddr_t dstaddr = *dstaddrp;
1212     uint16_t lport;
1213     int error;
1215     cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv4");
1217     ASSERT(connp->conn_ipversion == IPV4_VERSION);
1219     if (dstaddr == INADDR_ANY) {
1220         dstaddr = htonl(INADDR_LOOPBACK);
1221         *dstaddrp = dstaddr;
1222     }
1224     /* Handle __sin6_src_id if socket not bound to an IP address */
1225     if (srcid != 0 && connp->conn_laddr_v4 == INADDR_ANY) {
1226         ip_srcid_find_id(srcid, &connp->conn_laddr_v6,
1227                         IPCL_ZONEID(connp), dccps->dccps_netstack);
1228         connp->conn_saddr_v6 = connp->conn_laddr_v6;
1229     }
1231     IN6_IPADDR_TO_V4MAPPED(dstaddr, &connp->conn_faddr_v6);
1232     connp->conn_fport = dstport;
1234     if (dccp->dccp_state == DCCPS_CLOSED) {
1235         lport = dccp_update_next_port(dccps->dccps_next_port_to_try,
1236                                     dccp, B_TRUE);
1237         lport = dccp_bindi(dccp, lport, &connp->conn_laddr_v6, 0,
1238                           B_TRUE, B_FALSE, B_FALSE);
1239         if (lport == 0) {
1240             return (-TNOADDR);
1241         }
1242     }
1244     error = dccp_set_destination(dccp);
1245     if (error != 0) {
1246         return (error);
1247     }
1249     /*

```

```

1250     * Don't connect to oneself.
1251     */
1252     if (connp->conn_faddr_v4 == connp->conn_laddr_v4 &&
1253         connp->conn_fport == connp->conn_lport) {
1254         return (-TBADADDR);
1255     }
1257     dccp->dccp_state = DCCPS_REQUEST;
1259     return (ipcl_conn_insert_v4(connp));
1260 }
1262 /*
1263  * IPv6 connect.
1264  */
1265 static int
1266 dccp_connect_ipv6(dccp_t *dccp, in6_addr_t *dstaddrp, in_port_t dstport,
1267                  uint32_t flowinfo, uint_t srcid, uint32_t scope_id)
1268 {
1269     cmn_err(CE_NOTE, "dccp.c: dccp_connect_ipv6");
1271     return (0);
1272 }
1274 /*
1275  * Set the ports via conn_connect and build the template
1276  * header.
1277  */
1278 int
1279 dccp_set_destination(dccp_t *dccp)
1280 {
1281     conn_t *connp = dccp->dccp_connp;
1282     dccp_stack_t *dccps = dccp->dccp_dccps;
1283     iulp_t uinfo;
1284     uint32_t flags;
1285     int error;
1287     flags = IPDF_LSO | IPDF_ZCOPY;
1288     flags |= IPDF_UNIQUE_DCE;
1290     mutex_enter(&connp->conn_lock);
1291     error = conn_connect(connp, &uinfo, flags);
1292     mutex_exit(&connp->conn_lock);
1293     if (error != 0) {
1294         cmn_err(CE_NOTE, "conn_connect failed");
1295         return (error);
1296     }
1298     error = dccp_build_hdrs(dccp);
1299     if (error != 0) {
1300         cmn_err(CE_NOTE, "dccp_build_hdrs failed");
1301         return (error);
1302     }
1304     /* XXX */
1306     /* Initialise the ISS */
1307     dccp_iss_init(dccp);
1309     mutex_enter(&connp->conn_lock);
1310     connp->conn_state_flags &= ~CONN_INCIPIENT;
1311     mutex_exit(&connp->conn_lock);
1313     return (0);
1314 }

```

```
1316 /*
1317  * Clean up.
1318  */
1319 int
1320 dccp_clean_death(dccp_t *dccp, int error)
1321 {
1322     conn_t      *connp = dccp->dccp_connp;
1323     dccp_stack_t *dccps = dccp->dccp_dccps;
1324
1325     ASSERT(dccp != NULL);
1326     ASSERT((connp->conn_family == AF_INET &&
1327         connp->conn_ipversion == IPV4_VERSION) ||
1328         (connp->conn_family == AF_INET6 &&
1329         connp->conn_ipversion == IPV4_VERSION) ||
1330         (connp->conn_ipversion == IPV6_VERSION));
1331
1332     dccp_reinit(dccp);
1333     if (IPCL_IS_NONSTR(connp)) {
1334         (void) dccp_do_unbind(connp);
1335     }
1336
1337     return (-1);
1338 }
1339
1340 /*
1341  * Init the ISS.
1342  */
1343 static void
1344 dccp_iss_init(dccp_t *dccp)
1345 {
1346     cmn_err(CE_NOTE, "dccp.c: dccp_iss_init");
1347
1348     dccp->dccp_iss += gethrtime();
1349     dccp->dccp_gss = dccp->dccp_iss;
1350 }
1351
1352 /*
1353  * Reinitialization of a dccp structure.
1354  */
1355 static void
1356 dccp_reinit(dccp_t *dccp)
1357 {
1358     conn_t      *connp = dccp->dccp_connp;
1359     dccp_stack_t *dccps = dccp->dccp_dccps;
1360
1361     ASSERT(dccp->dccp_listener == NULL);
1362     ASSERT((connp->conn_family == AF_INET &&
1363         connp->conn_ipversion == IPV4_VERSION) ||
1364         (connp->conn_family == AF_INET6 &&
1365         connp->conn_ipversion == IPV4_VERSION) ||
1366         (connp->conn_ipversion == IPV6_VERSION));
1367
1368     /* Cancel outstanding timers */
1369     dccp_timers_stop(dccp);
1370 }
1371 #endif /* ! codereview */
```


new/usr/src/uts/common/inet/dccp/dccp.conf

1

913 Sat Aug 18 10:37:08 2012

new/usr/src/uts/common/inet/dccp/dccp.conf

dccp: starting module template

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License, Version 1.0 only
6 # (the "License"). You may not use this file except in compliance
7 # with the License.
8 #
9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 # or http://www.opensolaris.org/os/licensing.
11 # See the License for the specific language governing permissions
12 # and limitations under the License.
13 #
14 # When distributing Covered Code, include this CDDL HEADER in each
15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 # If applicable, add the following below this CDDL HEADER, with the
17 # fields enclosed by brackets "[]" replaced with your own identifying
18 # information: Portions Copyright [yyyy] [name of copyright owner]
19 #
20 # CDDL HEADER END
21 #
22 #
23 # Copyright (c) 1992, by Sun Microsystems, Inc.
24 #
25
26 name="dccp" parent="pseudo" instance=0;
27 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp6.conf

1

914 Sat Aug 18 10:37:09 2012

new/usr/src/uts/common/inet/dccp/dccp6.conf

dccp: clean up

```
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 Sat Aug 18 10:37:09 2012

new/usr/src/uts/common/inet/dccp/dccp6ddi.c

dccp: clean up

```
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

```
*****
17433 Sat Aug 18 10:37:09 2012
new/usr/src/uts/common/inet/dccp/dccp_bind.c
dccp: bind function
*****

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 Hoeppner. 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/squeue_impl.h>
46 #include <sys/squeue.h>
47 #include <sys/tsol/tnet.h>

49 #include <rpc/pmap_prot.h>

51 #include <inet/common.h>
52 #include <inet/dccp_impl.h>
53 #include <inet/ip.h>
54 #include <inet/proto_set.h>

56 #include <sys/cmn_err.h>

58 /* Setable in /etc/system */
59 static uint32_t dccp_random_anon_port = 1;

61 static int      dccp_bind_select_lport(dccp_t *, in_port_t *, boolean_t,
```

new/usr/src/uts/common/inet/dccp/dccp_bind.c

2

```
        cred_t *);
63 static in_port_t dccp_get_next_priv_port(const dccp_t *);

65 void
66 dccp_bind_hash_insert(dccp_df_t *tbf, dccp_t *dccp, int caller_holds_lock)
67 {
68     conn_t    *connp = dccp->dccp_connp;
69     conn_t    *connx;
70     dccp_t    **dccpp;
71     dccp_t    *dccpnext;
72     dccp_t    *dccphash;

74     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_hash_insert");

76     /* XXX:DCCP */

78     dccpp = tbf->df_dccp;
79     if (!caller_holds_lock) {
80         mutex_enter(&tbf->df_lock);
81     } else {
82         ASSERT(MUTEX_HELD(&tbf->df_lock));
83     }

85     dccphash = dccpp[0];
86     dccpnext = NULL;

88     if (dccphash != NULL) {
89         /* Look for an entry using the same port */
90         while ((dccphash = dccpp[0]) != NULL &&
91             connp->conn_lport != dccphash->dccp_connp->conn_lport) {
92             dccpp = &(dccphash->dccp_bind_hash);
93         }

95         /* The port was not found, just add to the end */
96         if (dccphash == NULL) {
97             goto insert;
98         }

100         dccpnext = dccphash;
101         connx = dccpnext->dccp_connp;
102         dccphash = NULL;
103         if (V6_OR_V4_INADDR_ANY(connp->conn_bound_addr_v6) &&
104             !V6_OR_V4_INADDR_ANY(connx->conn_bound_addr_v6)) {
105             while ((dccpnext = dccpp[0]) != NULL) {
106                 connx = dccpnext->dccp_connp;
107                 if (V6_OR_V4_INADDR_ANY(
108                     connx->conn_bound_addr_v6)) {
109                     dccpp = &(dccpnext->dccp_bind_hash_port)
110                 } else {
111                     break;
112                 }
113             }

115             if (dccpnext != NULL) {
116                 dccpnext->dccp_ptpbhn = &dccp->dccp_bind_hash_po
117                 dccphash = dccpnext->dccp_bind_hash;
118                 if (dccphash != NULL) {
119                     dccphash->dccp_ptpbhn =
120                         &(dccp->dccp_bind_hash);
121                     dccpnext->dccp_bind_hash = NULL;
122                 }
123             }
124         } else {
125             dccpnext->dccp_ptpbhn = &dccp->dccp_bind_hash_port;
126             dccphash = dccpnext->dccp_bind_hash;
127             if (dccphash != NULL) {
```

```

128         dccphash->dccp_ptpbhn =
129             &(dccp->dccp_bind_hash);
130         dccpNext->dccp_bind_hash = NULL;
131     }
132 }
133
134 insert:
135     dccp->dccp_bind_hash_port = dccpNext;
136     dccp->dccp_bind_hash = dccphash;
137     dccp->dccp_ptpbhn = dccp;
138     dccp[0] = dccp;
139
140     if (!caller_holds_lock) {
141         mutex_exit(&tbfd->df_lock);
142     }
143 }
144
145 /*
146  * Remove bind hash.
147  */
148 void
149 dccp_bind_hash_remove(dccp_t *dccp)
150 {
151     conn_t      *connp = dccp->dccp_connnp;
152     dccp_t      *dccpNext;
153     dccp_stack_t *dccps = dccp->dccp_dccps;
154     kmutex_t     *lockp;
155
156     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_hash_remove");
157
158     /* Nothing to remove */
159     if (dccp->dccp_ptpbhn == NULL) {
160         return;
161     }
162
163     ASSERT(connp->conn_lport != 0);
164     lockp = &dccps->dccps_bind_fanout[DCCP_BIND_HASH(connp->conn_lport,
165         dccps->dccps_bind_fanout_size)].df_lock;
166     ASSERT(lockp != NULL);
167
168     mutex_enter(lockp);
169     if (dccp->dccp_ptpbhn) {
170         dccpNext = dccp->dccp_bind_hash_port;
171         if (dccpNext != NULL) {
172             dccp->dccp_bind_hash_port = NULL;
173             dccpNext->dccp_ptpbhn = dccp->dccp_ptpbhn;
174             dccpNext->dccp_bind_hash = dccp->dccp_bind_hash;
175             if (dccpNext->dccp_bind_hash != NULL) {
176                 dccpNext->dccp_bind_hash->dccp_ptpbhn =
177                     &(dccpNext->dccp_bind_hash);
178                 dccp->dccp_bind_hash = NULL;
179             }
180         } else if ((dccpNext = dccp->dccp_bind_hash) != NULL) {
181             dccpNext->dccp_ptpbhn = dccp->dccp_ptpbhn;
182             dccp->dccp_bind_hash = NULL;
183         }
184         *dccp->dccp_ptpbhn = dccpNext;
185         dccp->dccp_ptpbhn = NULL;
186     }
187     mutex_exit(lockp);
188 }
189
190 /*
191  * Check for a valid address and get a local port.
192  */

```

```

194 int
195 dccp_bind_check(conn_t *connp, struct sockaddr *sa, socklen_t len, cred_t *cr,
196     boolean_t bind_to_req_port_only)
197 {
198     dccp_t      *dccp = connp->conn_dccp;
199     ip_stack_t   *ips = connp->conn_netstack->netstack_ip;
200     ip_xmit_attr_t *ixa = connp->conn_ixa;
201     sin_t        *sin;
202     sin6_t       *sin6;
203     ipaddr_t     v4addr;
204     in6_addr_t   v6addr;
205     ip_laddr_t   laddr_type = IPV4_UNICAST_UP;
206     zoneid_t     zoneid = IPCL_ZONEID(connp);
207     in_port_t     requested_port;
208     uint_t        scopeid = 0;
209     int           error;
210
211     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_check");
212
213     ASSERT((uintptr_t)len <= (uintptr_t)INT_MAX);
214
215     /*
216      * We should be in a pre-listen state.
217      */
218     if (dccp->dccp_state == DCCPS_LISTEN) {
219         return (0);
220     } else if (dccp->dccp_state > DCCPS_LISTEN) {
221         if (connp->conn_debug) {
222             (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
223                 "dccp_bind: bad state, %d", dccp->dccp_state);
224         }
225         return (-TOUTSTATE);
226     }
227
228     /*
229      * Check for a valid address parameter. Then validate the
230      * addresses and copy them and the required port in.
231      */
232     ASSERT(sa != NULL && len != 0);
233
234     if (!OK_32PTR((char *)sa)) {
235         if (connp->conn_debug) {
236             (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
237                 "dccp_bind: bad address parameter, "
238                 "address %p, len %d", (void *)sa, len);
239         }
240         return (-TPROTO);
241     }
242
243     error = proto_verify_ip_addr(connp->conn_family, sa, len);
244     if (error != 0) {
245         return (error);
246     }
247
248     switch (len) {
249     case sizeof (sin_t):
250         sin = (sin_t *)sa;
251         v4addr = sin->sin_addr.s_addr;
252         requested_port = ntohs(sin->sin_port);
253         IN6_IPADDR_TO_V4MAPPED(v4addr, &v6addr);
254         if (v4addr != INADDR_ANY) {
255             laddr_type = ip_laddr_verify_v4(v4addr, zoneid, ips,
256                 B_FALSE);
257         }
258         break;

```

```

260     case sizeof(sin6_t):
261         sin6 = (sin6_t *)sa;
262         v6addr = sin6->sin6_addr;
263         requested_port = ntohs(sin6->sin6_port);
264         if (IN6_IS_ADDR_V4MAPPED(&v6addr)) {
265             if (connp->conn_ipv6_v6only) {
266                 return (EADDRNOTAVAIL);
267             }
268
269             IN6_V4MAPPED_TO_IPADDR(&v6addr, v4addr);
270             if (v4addr != INADDR_ANY) {
271                 laddr_type = ip_laddr_verify_v4(v4addr, zoneid,
272                 ips, B_FALSE);
273             }
274         } else {
275             if (!IN6_IS_ADDR_UNSPECIFIED(&v6addr)) {
276                 if (IN6_IS_ADDR_LINKSCOPE(&v6addr)) {
277                     scopeid = sin6->sin6_scope_id;
278                     laddr_type = ip_laddr_verify_v6(&v6addr,
279                     zoneid, ips, B_FALSE, scopeid);
280                 }
281             }
282         }
283         break;
284
285     default:
286         if (connp->conn_debug) {
287             (void) strlog(DCCP_MOD_ID, 0, 1, SL_ERROR|SL_TRACE,
288             "dccp_bind: bad address length, %d", len);
289         }
290         return (EAFNOSUPPORT);
291 }
292
293 if (laddr_type == IPV4_BAD) {
294     return (EADDRNOTAVAIL);
295 }
296
297 connp->conn_bound_addr_v6 = v6addr;
298 if (scopeid != 0) {
299     ixa->ixa_flags |= IXAF_SCOPEID_SET;
300     ixa->ixa_scopeid = scopeid;
301     connp->conn_incoming_ifindex = scopeid;
302 } else {
303     ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
304     connp->conn_incoming_ifindex = connp->conn_bound_if;
305 }
306
307 connp->conn_laddr_v6 = v6addr;
308 connp->conn_saddr_v6 = v6addr;
309
310 bind_to_req_port_only = requested_port != 0 && bind_to_req_port_only;
311
312 error = dccp_bind_select_lport(dccp, &requested_port,
313 bind_to_req_port_only, cr);
314 if (error != 0) {
315     connp->conn_laddr_v6 = ipv6_all_zeros;
316     connp->conn_saddr_v6 = ipv6_all_zeros;
317     connp->conn_bound_addr_v6 = ipv6_all_zeros;
318 }
319
320 return (error);
321 }
322
323 /*
324  * Bind to a local port.
325  */

```

```

326 static int
327 dccp_bind_select_lport(dccp_t *dccp, in_port_t *requested_port_ptr,
328 boolean_t bind_to_req_port_only, cred_t *cr)
329 {
330     dccp_stack_t *dccps = dccp->dccp_dccps;
331     conn_t *connp = dccp->dccp_connp;
332     zone_t *zone;
333     in_port_t allocated_port;
334     in_port_t requested_port = *requested_port_ptr;
335     in_port_t mlp_port;
336     in6_addr_t v6addr = connp->conn_laddr_v6;
337     mlp_type_t addrtype;
338     mlp_type_t mlptype;
339     boolean_t user_specified;
340
341     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bind_select_lport");
342
343     ASSERT(cr != NULL);
344
345     mlptype = mlptSingle;
346     mlp_port = requested_port;
347     if (requested_port == 0) {
348         requested_port = connp->conn_anon_priv_bind ?
349         dccp_get_next_priv_port(dccp) :
350         dccp_update_next_port(dccps->dccps_next_port_to_try,
351         dccp, B_TRUE);
352     }
353     if (requested_port == 0) {
354         return (-TNOADDR);
355     }
356     user_specified = B_FALSE;
357
358     if (connp->conn_anon_mlp && is_system_labeled()) {
359         zone = crgetzone(cr);
360         addrtype = tsol_mlp_addr_type(
361         connp->conn_allzones ? ALL_ZONES : zone->zone_id,
362         IPV6_VERSION, &v6addr,
363         dccps->dccps_netstack->netstack_ip);
364         if (addrtype == mlptSingle) {
365             return (-TNOADDR);
366         }
367         mlptype = tsol_mlp_port_type(zone, IPPROTO_DCCP,
368         PMAPPORT, addrtype);
369         mlp_port = PMAPPORT;
370     } else {
371         int i;
372         boolean_t priv = B_FALSE;
373
374         if (requested_port < dccps->dccps_smallest_nonpriv_port) {
375             priv = B_TRUE;
376         } else {
377             for (i = 0; i < dccps->dccps_num_epriv_ports; i++) {
378                 if (requested_port ==
379                 dccps->dccps_epriv_ports[i]) {
380                     priv = B_TRUE;
381                     break;
382                 }
383             }
384         }
385
386         if (priv) {
387             if (secpolicy_net_privaddr(cr, requested_port,
388             IPPROTO_DCCP) != 0) {
389                 if (connp->conn_debug) {
390                     (void) strlog(DCCP_MOD_ID, 0, 1,
391                     SL_ERROR|SL_TRACE,

```

```

392         "tcp_bind: no priv for port %d",
393         requested_port);
394     }
395     return (-TACCES);
396 }
397
398 user_specified = B_TRUE;
399
400 connp = dccp->dccp_connp;
401 if (is_system_labeled()) {
402     zone = crgetzone(cr);
403     addrtype = tsol_mlp_addr_type(
404         connp->conn_allzones ? ALL_ZONES : zone->zone_id,
405         IPV6_VERSION, &v6addr,
406         dccps->dccps_netstack->netstack_ip);
407     if (addrtype == mlptSingle) {
408         return (-TNOADDR);
409     }
410     mlptype = tsol_mlp_port_type(zone, IPPROTO_DCCP,
411         requested_port, addrtype);
412 }
413
414 if (mlptype != mlptSingle) {
415     if (secpolicy_net_bindmlp(cr) != 0) {
416         if (connp->conn_debug) {
417             (void) strlog(DCCP_MOD_ID, 0, 1,
418                 SL_ERROR|SL_TRACE,
419                 "dccp_bind: no priv for multilevel port %d",
420                 requested_port);
421         }
422         return (-TACCES);
423     }
424 }
425
426 /*
427  * If we're specifically binding a shared IP address and the
428  * port is MLP on shared addresses, then check to see if this
429  * zone actually owns the MLP. Reject if not.
430  */
431 if (mlptype == mlptShared && addrtype == mlptShared) {
432     /*
433      * No need to handle exclusive-stack zones since
434      * ALL_ZONES only applies to the shared stack.
435      */
436     zoneid_t        mlpzone;
437
438     mlpzone = tsol_mlp_findzone(IPPROTO_DCCP,
439         htons(mlp_port));
440     if (connp->conn_zoneid != mlpzone) {
441         if (connp->conn_debug) {
442             (void) strlog(DCCP_MOD_ID, 0, 1,
443                 SL_ERROR|SL_TRACE,
444                 "dccp_bind: attempt to bind port "
445                 "%d on shared addr in zone %d "
446                 "(should be %d)",
447                 mlp_port, connp->conn_zoneid,
448                 mlpzone);
449         }
450         return (-TACCES);
451     }
452 }
453
454 if (!user_specified) {
455     int        error;

```

```

458     error = tsol_mlp_anon(zone, mlptype, connp->conn_proto,
459         requested_port, B_TRUE);
460     if (error != 0) {
461         if (connp->conn_debug) {
462             (void) strlog(DCCP_MOD_ID, 0, 1,
463                 SL_ERROR|SL_TRACE,
464                 "dccp_bind: cannot establish anon "
465                 "MLP for port %d",
466                 requested_port);
467         }
468         return (error);
469     }
470     connp->conn_anon_port = B_TRUE;
471 }
472 connp->conn_mlp_type = mlptype;
473
474 allocated_port = dccp_bindi(dccp, requested_port, &v6addr,
475     connp->conn_reuseaddr, B_FALSE, bind_to_req_port_only,
476     user_specified);
477
478 if (allocated_port == 0) {
479     connp->conn_mlp_type = mlptSingle;
480
481     if (connp->conn_anon_port) {
482         connp->conn_anon_port = B_FALSE;
483         (void) tsol_mlp_anon(zone, mlptype, connp->conn_proto,
484             requested_port, B_FALSE);
485     }
486
487     if (bind_to_req_port_only) {
488         if (connp->conn_debug) {
489             (void) strlog(DCCP_MOD_ID, 0, 1,
490                 SL_ERROR|SL_TRACE,
491                 "dccp_bind: requested addr busy");
492         }
493         return (-TADDRBUSY);
494     } else {
495         /* If we are out of ports, fail the bind */
496         if (connp->conn_debug) {
497             (void) strlog(DCCP_MOD_ID, 0, 1,
498                 SL_ERROR|SL_TRACE,
499                 "dccp_bind: out of ports?");
500         }
501         return (-TNOADDR);
502     }
503 }
504
505 /* Pass the allocated port back */
506 *requested_port_ptr = allocated_port;
507 return (0);
508 }
509
510 in_port_t
511 dccp_bindi(dccp_t *dccp, in_port_t port, const in6_addr_t *laddr,
512     int reuseaddr, boolean_t quick_connect, boolean_t bind_to_req_port_only,
513     boolean_t user_specified)
514 {
515     dccp_stack_t    *dccps = dccp->dccp_dccps;
516     conn_t          *connp = dccp->dccp_connp;
517     int              count = 0;
518     int              loopmax;
519
520     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bindi");
521
522     if (bind_to_req_port_only) {

```

```

524     loopmax = 1;
525 } else {
526     if (connp->conn_anon_priv_bind) {
527         loopmax = IPPORT_RESERVED -
528             dccps->dccps_min_anonpriv_port;
529     } else {
530         loopmax = (dccps->dccps_largest_anon_port -
531             dccps->dccps_smallest_anon_port + 1);
532     }
533 }

535 do {
536     conn_t      *lconnp;
537     dccp_t      *ldccp;
538     dccp_df_t    *ldf;
539     uint16_t     lport;

541     lport = htons(port);

543     dccp_bind_hash_remove(dccp);
544     ldf = &dccps->dccps_bind_fanout[DCCP_BIND_HASH(lport,
545         dccps->dccps_bind_fanout_size)];
546     mutex_enter(&ldf->df_lock);
547     for (ldccp = ldf->df_dccp; ldccp != NULL;
548         ldccp = ldccp->dccp_bind_hash) {
549         if (lport == ldccp->dccp_connp->conn_lport) {
550             break;
551         }
552     }

554     if (ldccp != NULL) {
555         /* The port number is busy */
556         mutex_exit(&ldf->df_lock);
557     } else {
558         /*
559          * This port is ours. Insert in fanout and mark as
560          * bound to prevent others from getting the port
561          * number.
562          */
563         dccp->dccp_state = DCCPS_BOUND;
564         DTRACE_DCCP6(state_change, void, NULL,
565             ip_xmit_attr_t *, connp->conn_ixa,
566             void, NULL, dccp_t *, dccp, void, NULL,
567             int32_t, DCCPS_CLOSED);

569         connp->conn_lport = htons(port);

571         ASSERT(&dccps->dccps_bind_fanout[DCCP_BIND_HASH(
572             connp->conn_lport,
573             dccps->dccps_bind_fanout_size)] == ldf);
574         dccp_bind_hash_insert(ldf, dccp, 1);

576         mutex_exit(&ldf->df_lock);

578         /*
579          * We don't want tcp_next_port_to_try to "inherit"
580          * a port number supplied by the user in a bind.
581          */
582         if (user_specified) {
583             return (port);
584         }

586         /*
587          * This is the only place where dccp_next_port_to_try
588          * is updated. After the update, it may or may not
589          * be in the valid range.

```

```

590         /*
591         if (!connp->conn_anon_priv_bind) {
592             dccps->dccps_next_port_to_try = port + 1;
593         }

595         return (port);
596     }

598     if (connp->conn_anon_priv_bind) {
599         port = dccp_get_next_priv_port(dccp);
600     } else {
601         if (count == 0 && user_specified) {
602             /*
603              * We may have to return an anonymous port. So
604              * get one to start with.
605              */
606             port =
607                 dccp_update_next_port(
608                     dccps->dccps_next_port_to_try,
609                     dccp, B_TRUE);
610             user_specified = B_FALSE;
611         } else {
612             port = dccp_update_next_port(port + 1, dccp,
613                 B_FALSE);
614         }
615     }

617     if (port == 0) {
618         break;
619     }

621 } while (++count < loopmax);

623     cmn_err(CE_NOTE, "dccp_bind.c: dccp_bindi exit");

625     return (0);
626 }

628 in_port_t
629 dccp_update_next_port(in_port_t port, const dccp_t *dccp, boolean_t random)
630 {
631     dccp_stack_t *dccps = dccp->dccp_dccps;
632     boolean_t restart = B_FALSE;
633     int i;

635     cmn_err(CE_NOTE, "dccp_bind.c: dccp_update_next_port");

637     if (random && dccp_random_anon_port != 0) {
638         (void) random_get_pseudo_bytes((uint8_t *)&port,
639             sizeof (in_port_t));

641         if (port < dccps->dccps_smallest_anon_port) {
642             port = dccps->dccps_smallest_anon_port +
643                 port % (dccps->dccps_largest_anon_port -
644                     dccps->dccps_smallest_anon_port);
645         }
646     }

648 retry:
649     if (port < dccps->dccps_smallest_anon_port) {
650         port = (in_port_t)dccps->dccps_smallest_anon_port;
651     }

653     if (port > dccps->dccps_largest_anon_port) {
654         if (restart) {
655             return (0);

```



```
656     }
657     restart = B_TRUE;
658     port = (in_port_t) dccps->dccps_smallest_anon_port;
659 }
661 if (port < dccps->dccps_smallest_nonpriv_port) {
662     port = (in_port_t) dccps->dccps_smallest_nonpriv_port;
663 }
665 for (i = 0; i < dccps->dccps_num_epriv_ports; i++) {
666     if (port == dccps->dccps_epriv_ports[i]) {
667         port++;
668         goto retry;
669     }
670 }
672 return (port);
673 }
675 /*
676  * Return the next anonymous port in the privileged port range for
677  * bind checking. It starts at IPPORT_RESERVED - 1 and goes
678  * downwards. This is the same behavior as documented in the userland
679  * library call rresvport(3N).
680  *
681  * TS note: skip multilevel ports.
682  */
683 static in_port_t
684 dccp_get_next_priv_port(const dccp_t *dccp)
685 {
686     static in_port_t next_priv_port = IPPORT_RESERVED - 1;
687     dccp_stack_t *dccps = dccp->dccp_dccps;
688     in_port_t nextport;
689     boolean_t restart = B_FALSE;
691 retry:
692     if (next_priv_port < dccps->dccps_min_anonpriv_port ||
693         next_priv_port >= IPPORT_RESERVED) {
694         next_priv_port = IPPORT_RESERVED - 1;
695         if (restart) {
696             return (0);
697         }
698         restart = B_TRUE;
699     }
701     if (is_system_labeled() &&
702         (nextport = tsol_next_port(crgetzone(dccp->dccp_connp->conn_cred),
703             next_priv_port, IPPROTO_DCCP, B_FALSE)) != 0) {
704         next_priv_port = nextport;
705         goto retry;
706     }
708     return (next_priv_port--);
709 }
710 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_features.c

1

2878 Sat Aug 18 10:37:09 2012
new/usr/src/uts/common/inet/dccp/dccp_features.c
dccp: options and features

```
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/stream.h>
18 #include <sys/debug.h>
19 #include <sys/cmn_err.h>
20 #include <inet/dccp_impl.h>
21 #include <inet/dccp_stack.h>

23 /*
24  * This file contains functions to parse and process DCCP features.
25 */

27 /*
28  * Feature handling.
29 */
30 int
31 dccp_parse_feature(dccp_t *dccp, uint8_t option, uint8_t length, uchar_t *up,
32                  boolean_t mandatory)
33 {
34     dccp_feature_t *feature;
35     uint64_t feature_value;
36     uint8_t feature_type;
37     uint8_t feature_length = length - 1;

39     cmn_err(CE_NOTE, "dccp_features.c: dccp_parse_feature");

41     feature_type = *up;
42     up++;

44     switch (feature_type) {
45     case DCCP_FEATURE_CCID:
46         cmn_err(CE_NOTE, "DCCP_FEATURE_CCID");
47         break;
48     case DCCP_FEATURE_ALLOW_SHORT_SEQNOS:
49         cmn_err(CE_NOTE, "DCCP_FEATURE_ALLOW_SHORT_SEQNOS");
50         break;
51     case DCCP_FEATURE_SEQUENCE_WINDOW:
52         cmn_err(CE_NOTE, "DCCP_FEATURE_SEQUENCE_WINDOW");
53         if (feature_length != 6)
54             return (DCCP_RESET_MANDATORY_ERROR);

56         feature_value = (uint64_t)up[0] << 40;
57         feature_value += ((uint64_t)up[1] << 32);
58         feature_value += ((uint64_t)up[2] << 24);
59         feature_value += ((uint64_t)up[3] << 16);
60         feature_value += ((uint64_t)up[4] << 8);
61         feature_value += (uint64_t)up[5];
```

new/usr/src/uts/common/inet/dccp/dccp_features.c

2

```
63         if (feature_value < 32 || feature_value > 70368744177663)
64             return (DCCP_RESET_MANDATORY_ERROR);
65         dccp->dccp_sequence_window = feature_value;
66         break;
67     case DCCP_FEATURE_ECN_INCAPABLE:
68         cmn_err(CE_NOTE, "DCCP_FEATURE_ECN_INCAPABLE");
69         break;
70     case DCCP_FEATURE_ACK_RATIO:
71         cmn_err(CE_NOTE, "DCCP_FEATURE_ACK_RATIO");
72         break;
73     case DCCP_FEATURE_SEND_ACK_VECTOR:
74         cmn_err(CE_NOTE, "DCCP_FEATURE_SEND_ACK_VECTOR");
75         break;
76     case DCCP_FEATURE_SEND_NDP_COUNT:
77         cmn_err(CE_NOTE, "DCCP_FEATURE_SEND_NDP_COUNT");
78         break;
79     case DCCP_FEATURE_MIN_CHECKSUM_COVERAGE:
80         cmn_err(CE_NOTE, "DCCP_FEATURE_MIN_CHECKSUM_COVERAGE");
81         break;
82     case DCCP_FEATURE_CHECK_DATA_CHECKSUM:
83         cmn_err(CE_NOTE, "DCCP_FEATURE_CHECK_DATA_CHECKSUM");
84         break;
85     default:
86         cmn_err(CE_NOTE, "ERROR DEFAULT");
87         break;
88     }

90     cmn_err(CE_NOTE, "feature len: %d", feature_length);

92     feature = (dccp_feature_t *)kmem_alloc(sizeof (dccp_feature_t),
93                                             KM_SLEEP);
94     if (feature == NULL) {
95         return (0);
96     }

98     feature->df_option = option;
99     feature->df_type = feature_type;
100    feature->df_mandatory = mandatory;

102    list_insert_tail(&dccp->dccp_features, feature);

104    return (0);
105 }

107 void
108 dccp_features_init(void)
109 {
110 }

112 void
113 dccp_features_destroy(void)
114 {
115 }
116 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_input.c

1

```
*****
25987 Sat Aug 18 10:37:10 2012
new/usr/src/uts/common/inet/dccp/dccp_input.c
dccp: starting module template
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

27 /*
28  * Copyright 2012 David Hoeppner. 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/tinet.h>

45 #include <inet/common.h>
46 #include <inet/dccp_impl.h>
47 #include <inet/ip.h>

49 #include <sys/cmn_err.h>

51 static mblk_t *dccp_conn_create_v4(conn_t *, conn_t *, mblk_t *,
52 ip_recv_attr_t *);
53 static mblk_t *dccp_conn_create_v6(conn_t *, conn_t *, mblk_t *,
54 ip_recv_attr_t *);
55 static void dccp_input_listener(void *, mblk_t *, void *, ip_recv_attr_t *);
56 static void dccp_icmp_error_ipv6(dccp_t *, mblk_t *, ip_recv_attr_t *);
57 static int dccp_check_input(conn_t *, mblk_t *, ip_recv_attr_t *);

59 void
60 dccp_icmp_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
61 {
```

new/usr/src/uts/common/inet/dccp/dccp_input.c

2

```
62 conn_t *connp = (conn_t *)arg1;
63 dccp_t *dccp = connp->conn_dccp;
64 ipha_t *ipha;
65 icmph_t *icmph;
66 sin_t sin;
67 sin6_t sin6;
68 int iph_hdr_length;
69 int error;

71 cmn_err(CE_NOTE, "dccp_input.c: dccp_icmp_input");

73 ipha = (ipha_t *)mp->b_rptr;

75 /* Assume IP provides aligned packets */
76 ASSERT(OK_32PTR(mp->b_rptr));
77 ASSERT((MBLKL(mp) >= sizeof (ipha_t)));

79 if (dccp->dccp_state == DCCPS_CLOSED) {
80     freemsg(mp);
81     return;
82 }

84 /*
85  * Verify IP version. Anything other than IPv4 or IPv6 packet is sent
86  * upstream. ICMPv6 is handled in dccp_icmp_error_ipv6.
87  */
88 if (IPH_HDR_VERSION(ipha) != IPV4_VERSION) {
89     ASSERT(IPH_HDR_VERSION(ipha) == IPV6_VERSION);
90     dccp_icmp_error_ipv6(dccp, mp, ira);
91     return;
92 }
93 ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);

95 ASSERT(IPH_HDR_LENGTH(ipha) == ira->ira_ip_hdr_length);
96 /* Skip past the outer IP and ICMP headers */
97 iph_hdr_length = ira->ira_ip_hdr_length;
98 icmph = (icmph_t *)&mp->b_rptr[iph_hdr_length];

100 switch (icmph->icmph_type) {
101     case ICMP_DEST_UNREACHABLE:
102         switch (icmph->icmph_code) {
103             case ICMP_FRAGMENTATION_NEEDED:
104             case ICMP_PORT_UNREACHABLE:
105             case ICMP_PROTOCOL_UNREACHABLE:
106             case ICMP_HOST_UNREACHABLE:
107             case ICMP_NET_UNREACHABLE:
108                 break;
109             default:
110                 break;
111         }
112         break;
113     case ICMP_SOURCE_QUENCH:
114         break;
115 }

117 freemsg(mp);
118 }

120 /*
121  * Handler for ICMPv6 error messages.
122  */
123 static void
124 dccp_icmp_error_ipv6(dccp_t *dccp, mblk_t *mp, ip_recv_attr_t *ira)
125 {
126     ip6_t *ip6h;
127     icmp6_t *icmp6;
```

```

128     uint16_t      iph_hdr_length = ira->ira_ip_hdr_length;
130
131     cmn_err(CE_NOTE, "dccp_input.c: dccp_icmp_error_ip6");
132
133     ASSERT((MBLKL(mp) >= sizeof (ip6_t)));
134
135     icmp6 = (icmp6_t *) &mp->b_rptr[iph_hdr_length];
136     ip6h = (ip6_t *) &icmp6[1];
137 }
138
139 void
140 dccp_rsrv(queue_t *q)
141 {
142     cmn_err(CE_NOTE, "dccp_input.c: dccp_rsrv");
143 }
144
145 /*
146  * Handle a REQUEST on an AF_INET6 socket; can be either IPv4 or IPv6.
147  */
148 static mblk_t *
149 dccp_conn_create_v6(conn_t *lconnp, conn_t *connp, mblk_t *mp,
150     ip_rcv_attr_t *ira)
151 {
152     dccp_t      *ldccp = lconnp->conn_dccp;
153     dccp_t      *dccp = connp->conn_dccp;
154     dccp_stack_t *dccps = dccp->dccp_dccps;
155     ipha_t      *ipha;
156     ip6_t      *ip6h;
157     mblk_t      *tpi_mp;
158     sin6_t      sin6;
159     uint_t      ifindex = ira->ira_ruifindex;
160
161     if (ira->ira_flags & IRAF_IS_IPV4) {
162         ipha = (ipha_t *) mp->b_rptr;
163
164         connp->conn_ipversion = IPV4_VERSION;
165         IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &connp->conn_laddr_v6);
166         IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &connp->conn_faddr_v6);
167         connp->conn_saddr_v6 = connp->conn_laddr_v6;
168
169         sin6 = sin6_null;
170         sin6.sin6_addr = connp->conn_faddr_v6;
171         sin6.sin6_port = connp->conn_fport;
172         sin6.sin6_family = AF_INET6;
173         sin6._sin6_src_id = ip_srcid_find_addr(&connp->conn_laddr_v6,
174             IPL_ZONEID(lconnp), dccps->dccps_netstack);
175
176         if (connp->conn_rcv_ancillary.crb_recvdstaddr) {
177             sin6_t sin6d;
178
179             sin6d = sin6_null;
180             sin6d.sin6_addr = connp->conn_laddr_v6;
181             sin6d.sin6_port = connp->conn_lport;
182             sin6d.sin6_family = AF_INET;
183             tpi_mp = mi_tpi_extconn_ind(NULL,
184                 (char *)&sin6d, sizeof (sin6_t),
185                 (char *)&dccp,
186                 (t_scalar_t) sizeof (intptr_t),
187                 (char *)&sin6d, sizeof (sin6_t),
188                 (t_scalar_t) ldccp->dccp_conn_req_seqnum);
189         } else {
190             tpi_mp = mi_tpi_conn_ind(NULL,
191                 (char *)&sin6, sizeof (sin6_t),
192                 (char *)&dccp, (t_scalar_t) sizeof (intptr_t),
193                 (t_scalar_t) ldccp->dccp_conn_req_seqnum);

```

```

194     }
195     } else {
196         ip6h = (ip6_t *) mp->b_rptr;
197
198         connp->conn_ipversion = IPV6_VERSION;
199         connp->conn_laddr_v6 = ip6h->ip6_dst;
200         connp->conn_faddr_v6 = ip6h->ip6_src;
201         connp->conn_saddr_v6 = connp->conn_laddr_v6;
202
203         sin6 = sin6_null;
204         sin6.sin6_addr = connp->conn_faddr_v6;
205         sin6.sin6_port = connp->conn_fport;
206         sin6.sin6_family = AF_INET6;
207         sin6.sin6_flowinfo = ip6h->ip6_vcf & ~IPV6_VERS_AND_FLOW_MASK;
208         sin6._sin6_src_id = ip_srcid_find_addr(&connp->conn_laddr_v6,
209             IPL_ZONEID(lconnp), dccps->dccps_netstack);
210
211         if (IN6_IS_ADDR_LINKSCOPE(&ip6h->ip6_src)) {
212             /* Pass up the scope_id of remote addr */
213             sin6.sin6_scope_id = ifindex;
214         } else {
215             sin6.sin6_scope_id = 0;
216         }
217         if (connp->conn_rcv_ancillary.crb_recvdstaddr) {
218             sin6_t sin6d;
219
220             sin6d = sin6_null;
221             sin6d.sin6_addr = connp->conn_laddr_v6;
222             sin6d.sin6_port = connp->conn_lport;
223             sin6d.sin6_family = AF_INET6;
224             if (IN6_IS_ADDR_LINKSCOPE(&connp->conn_laddr_v6))
225                 sin6d.sin6_scope_id = ifindex;
226
227             tpi_mp = mi_tpi_extconn_ind(NULL,
228                 (char *)&sin6d, sizeof (sin6_t),
229                 (char *)&dccp, (t_scalar_t) sizeof (intptr_t),
230                 (char *)&sin6d, sizeof (sin6_t),
231                 (t_scalar_t) ldccp->dccp_conn_req_seqnum);
232         } else {
233             tpi_mp = mi_tpi_conn_ind(NULL,
234                 (char *)&sin6, sizeof (sin6_t),
235                 (char *)&dccp, (t_scalar_t) sizeof (intptr_t),
236                 (t_scalar_t) ldccp->dccp_conn_req_seqnum);
237         }
238     }
239
240     /* XXX mss */
241     return (tpi_mp);
242 }
243
244 /*
245  * Handle a REQUEST on an AF_INET socket.
246  */
247 static mblk_t *
248 dccp_conn_create_v4(conn_t *lconnp, conn_t *connp, mblk_t *mp,
249     ip_rcv_attr_t *ira)
250 {
251     dccp_t      *ldccp = lconnp->conn_dccp;
252     dccp_t      *dccp = connp->conn_dccp;
253     dccp_stack_t *dccps = dccp->dccp_dccps;
254     ipha_t      *ipha;
255     mblk_t      *tpi_mp;
256     sin_t      sin;
257
258     ASSERT(ira->ira_flags & IRAF_IS_IPV4);
259     ipha = (ipha_t *) mp->b_rptr;

```

```

261     connp->conn_ipversion = IPV4_VERSION;
262     IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &connp->conn_laddr_v6);
263     IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &connp->conn_faddr_v6);
264     connp->conn_saddr_v6 = connp->conn_laddr_v6;

266     sin = sin_null;
267     sin.sin_addr.s_addr = connp->conn_faddr_v4;
268     sin.sin_port = connp->conn_fport;
269     sin.sin_family = AF_INET;

271     if (lconnp->conn_rcv_ancillary.crb_rcvdstaddr) {
272         cmn_err(CE_NOTE, "ancillary");

274         sin_t    sind;

276         sind = sin_null;
277         sind.sin_addr.s_addr = connp->conn_laddr_v4;
278         sind.sin_port = connp->conn_lport;
279         sind.sin_family = AF_INET;

281         tpi_mp = mi_tpi_extconn_ind(NULL,
282             (char *)&sind, sizeof (sin_t), (char *)&dccp,
283             (t_scalar_t)sizeof (intptr_t), (char *)&sind,
284             sizeof (sin_t), (t_scalar_t)ldccp->dccp_conn_req_seqnum);

286     } else {
287         tpi_mp = mi_tpi_conn_ind(NULL,
288             (char *)&sin, sizeof (sin_t),
289             (char *)&dccp, (t_scalar_t)sizeof (intptr_t),
290             (t_scalar_t)ldccp->dccp_conn_req_seqnum);
291     }

293     /* XXX mss */

295     return (tpi_mp);
296 }

298 static void
299 dccp_input_listener(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *ira)
300 {
301     conn_t      *lconnp = (conn_t *)arg;
302     conn_t      *econnp;
303     dccp_t      *listener = lconnp->conn_dccp;
304     dccp_t      *eager;
305     dccp_stack_t *dccps = listener->dccp_dccps;
306     ip_stack_t  *ipst = dccps->dccps_netstack->netstack_ip;
307     dccpha_t    *dccpha;
308     dccpha_ext_t *dccpha_ext;
309     squeue_t    *new_sq;
310     mblk_t      *tpi_mp;
311     mblk_t      *mpl;
312     uint64_t    seq;
313     uint_t      ifindex = ira->ira_ruifindex;
314     uint_t      ip_hdr_len;
315     uint_t      type;
316     int         error;

318     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener");

321     DTRACE_DCCP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, lconnp->conn_ixa
322     _dtrace_dccp_void_ip_t *, mp->b_rptr, dccp_t *, listener,
323     _dtrace_dccp_dccph_t *, dccpha);

325     /* Basic checks on packet */

```

```

326     if (dccp_check_input(lconnp, mp, ira) == 0) {
327         cmn_err(CE_NOTE, "dccp_check_input failed.");
328         freemsg(mp);
329         return;
330     }

332     ip_hdr_len = ira->ira_ip_hdr_length;
333     dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];
334     dccpha_ext = (dccpha_ext_t *)&mp->b_rptr[ip_hdr_len + sizeof (dccpha_t)]
335     type = (uint_t)dccpha->dha_type;

337     if (type != DCCP_PKT_REQUEST) {
338         if (type != DCCP_PKT_RESET) {
339             listener->dccp_reset_code = DCCP_RESET_NO_CONNECTION;
340             dccp_xmit_listeners_reset(mp, ira, ipst, lconnp);
341         }

343         freemsg(mp);
344         return;
345     }

347     /* XXX memory pressure */

349     /* XXX request defense */

351     /* XXX number of connections per listener */

353     ASSERT(ira->ira_sq == NULL);
354     new_sq = ira->ira_sq;

356     econnp = (conn_t *)dccp_get_conn(arg2, dccps);
357     if (econnp == NULL) {
358         cmn_err(CE_NOTE, "econnp not found (eager)");
359         goto error2;
360     }

362     ASSERT(econnp->conn_netstack == lconnp->conn_netstack);
363     econnp->conn_sq = new_sq;
364     econnp->conn_initial_sq = new_sq;
365     econnp->conn_ixa->ixa_sq = new_sq;

367     econnp->conn_fport = dccpha->dha_lport;
368     econnp->conn_lport = dccpha->dha_fport;

370     error = conn_inherit_parent(lconnp, econnp);
371     if (error != 0) {
372         cmn_err(CE_NOTE, "conn_inherit_parent failed");
373         goto error3;
374     }

376     /* We already know the laddr of the new connection is ours */
377     econnp->conn_ixa->ixa_src_generation = ipst->ips_src_generation;

379     ASSERT(OK_32PTR(mp->b_rptr));
380     ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION ||
381         IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);

383     if (lconnp->conn_family == AF_INET) {
384         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION);
385         tpi_mp = dccp_conn_create_v4(lconnp, econnp, mp, ira);
386     } else {
387         tpi_mp = dccp_conn_create_v6(lconnp, econnp, mp, ira);
388     }

390     if (tpi_mp == NULL) {
391         cmn_err(CE_NOTE, "tpi_mo == NULL");

```

```

392         goto error3;
393     }

395     eager = econnp->conn_dccp;
396     SOCK_CONNID_INIT(eager->dccp_connid);

398     dccp_init_values(eager, listener);

400     ASSERT((econnp->conn_ixa->ixa_flags &
401             (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
402             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO)) ==
403            (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
404            IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO));

406     if (!(ira->ira_flags & IRAF_IS_IPV4) && econnp->conn_bound_if == 0) {
407         if (IN6_IS_ADDR_LINKSCOPE(&econnp->conn_faddr_v6) ||
408             IN6_IS_ADDR_LINKSCOPE(&econnp->conn_laddr_v6)) {
409             econnp->conn_incoming_ifindex = ifindex;
410             econnp->conn_ixa->ixa_flags |= IXAF_SCOPEID_SET;
411             econnp->conn_ixa->ixa_scopeid = ifindex;
412         }
413     }

415     if (ira->ira_cred != NULL) {
416         mblk_setcred(tpi_mp, ira->ira_cred, ira->ira_cpid);
417     }

419     if (IPCL_IS_NONSTR(lconnp)) {
420         econnp->conn_flags |= IPCL_NONSTR;
421     }

423     /* XXX dccps is right? */
424     dccp_bind_hash_insert(&dccps->dccps_bind_fanout[
425         DCCP_BIND_HASH(econnp->conn_lport, dccps->dccps_bind_fanout_size)],
426         eager, 0);

428     /* XXX CLUSTER */

430     SOCK_CONNID_BUMP(eager->dccp_connid);

432     error = dccp_set_destination(eager);
433     if (error != 0) {
434         cmn_err(CE_NOTE, "dccp_set_destination failed.");
435         DCCPS_BUMP_MIB(dccps, dccpAttemptFails);
436         /* Undo the bind_hash_insert */
437         dccp_bind_hash_remove(eager);
438         goto error3;
439     }

441     /* Eager lock */
442     CONN_INC_REF(lconnp);

444     /* Set tcp_listener before adding it to tcp_conn_fanout */
445     eager->dccp_listener = listener;
446     eager->dccp_saved_listener = listener;

448     eager->dccp_conn_req_seqnum = listener->dccp_conn_req_seqnum;
449     if (++listener->dccp_conn_req_seqnum == -1) {
450         /*
451          * -1 is "special" and defined in TPI as something
452          * that should never be used in T_CONN_IND
453          */
454         ++listener->dccp_conn_req_seqnum;
455     }

457     /* XXX SYN DEFENSE */

```

```

459     seq = ntohs(dccpha->dha_seq);
460     seq = seq << 32;
461     seq |= ntohl(dccpha_ext->dha_ext_seq);

463     /* ISS was set in set_destination */
464     eager->dccp_gar = eager->dccp_iss;
465     eager->dccp_gsr = seq;
466     eager->dccp_osr = seq;

468     /* Process all DCCP options */
469     error = dccp_process_options(eager, dccpha);
470     if (error != 0) {
471         cmn_err(CE_NOTE, "dccp_process_options failed.");
472         mpl = dccp_generate_reset(econnp);
473         eager->dccp_state = DCCPS_CLOSED;
474         if (mpl == NULL) {
475             cmn_err(CE_NOTE, "dccp_generate_reset failed.");
476             goto error;
477         }

479         ip_xmit_attr_t *ixa = econnp->conn_ixa;

481         ixa->ixa_pktlen = msgdsize(mpl);
482         (void) conn_ip_output(mpl, ixa);

484         return;
485     }

487     eager->dccp_state = DCCPS_RESPOND;
488     DTRACE_DCCP6(state_change, void, NULL, ip_xmit_attr_t *,
489         econnp->conn_ixa, void, NULL, dccp_t *, eager, void, NULL,
490         int32_t, DCCPS_LISTEN);

492     mpl = dccp_generate_response(econnp, mp);
493     if (mpl == NULL) {
494         cmn_err(CE_NOTE, "dccp_generate_packet failed");
495         /*
496          * Increment the ref count as we are going to
497          * enqueueing an mp in queue
498          */
499         CONN_INC_REF(econnp);
500         goto error;
501     }

503     CONN_INC_REF(econnp);

505     error = ipcl_conn_insert(econnp);
506     if (error != 0) {
507         cmn_err(CE_NOTE, "ipcl_conn_insert(econnp) failed");
508         goto error;
509     }

511     ASSERT(econnp->conn_ixa->ixa_notify_cookie == econnp->conn_dccp);
512     freemsg(mp);

514     /*
515      * Send the RESPONSE. Use the right squeue so that conn_ixa is
516      * only used by one thread at a time.
517      */
518     if (econnp->conn_sqp == lconnp->conn_sqp) {
519         DTRACE_DCCP5(send, mblk_t *, NULL, ip_xmit_attr_t *,
520             econnp->conn_ixa, __dtrace_dccp_void_ip_t *, mpl->b_rptr,
521             dccp_t *, eager, __dtrace_dccp_dccph_t *,
522             &mpl->b_rptr[econnp->conn_ixa->ixa_ip_hdr_length]);
523         (void) conn_ip_output(mpl, econnp->conn_ixa);

```

```

524         CONN_DEC_REF(econnp);
525     } else {
526         SQUEUE_ENTER_ONE(econnp->conn_sq, mp1, dccp_send_synack,
527             econnp, NULL, SQ_PROCESS, SQTAG_DCCP_SEND_RESPONSE);
528     }
529
530     return;
531 error:
532     freemsg(mp1);
533 error2:
534     CONN_DEC_REF(econnp);
535 error3:
536     freemsg(mp);
537 }
538
539 void
540 dccp_input_listener_unbound(void *arg, mblk_t *mp, void *arg2,
541     ip_rcv_attr_t *ira)
542 {
543     conn_t      *connp = (conn_t *)arg;
544     squeue_t     *sq = (squeue_t *)arg2;
545     squeue_t     *new_sq;
546     uint32_t      conn_flags;
547
548     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_listener_unbound");
549
550     ASSERT(ira->ira_sq != NULL);
551     new_sq = ira->ira_sq;
552
553     if (connp->conn_fanout == NULL) {
554         goto done;
555     }
556
557     /*
558      * Bind to correct queue.
559      */
560     if (!(connp->conn_flags & IPCL_FULLY_BOUND)) {
561         cmn_err(CE_NOTE, "not fully bound");
562
563         mutex_enter(&connp->conn_fanout->connf_lock);
564         mutex_enter(&connp->conn_lock);
565
566         if (connp->conn_ref != 4 ||
567             connp->conn_dccp->dccp_state != DCCPS_LISTEN) {
568             mutex_exit(&connp->conn_lock);
569             mutex_exit(&connp->conn_fanout->connf_lock);
570             goto done;
571         }
572
573         if (connp->conn_sq != new_sq) {
574             while (connp->conn_sq != new_sq) {
575                 (void) casptr(&connp->conn_sq, sq, new_sq);
576             }
577             connp->conn_ixa->ixa_sq = new_sq;
578         }
579
580         do {
581             conn_flags = connp->conn_flags;
582             conn_flags |= IPCL_FULLY_BOUND;
583             (void) cas32(&connp->conn_flags, connp->conn_flags,
584                 conn_flags);
585         } while (!(connp->conn_flags & IPCL_FULLY_BOUND));
586
587         mutex_exit(&connp->conn_lock);
588         mutex_exit(&connp->conn_fanout->connf_lock);

```

```

590         connp->conn_rcv = dccp_input_listener;
591     }
592
593 done:
594     if (connp->conn_sq != sq) {
595         CONN_INC_REF(connp);
596         SQUEUE_ENTER_ONE(connp->conn_sq, mp, connp->conn_rcv, connp,
597             ira, SQ_FILL, SQTAG_DCCP_CONN_REQ_UNBOUND);
598     } else {
599         dccp_input_listener(connp, mp, sq, ira);
600     }
601 }
602
603 boolean_t
604 dccp_verifyicmp(conn_t *connp, void *arg2, icmph_t *icmph, icmp6_t *icmp6,
605     ip_rcv_attr_t *ira)
606 {
607     cmn_err(CE_NOTE, "dccp_input.c: dccp_verifyicmp");
608
609     return (B_TRUE);
610 }
611
612 /*
613  * Basic sanity checks on all input packets.
614  */
615 static int
616 dccp_check_input(conn_t *connp, mblk_t *mp, ip_rcv_attr_t *ira)
617 {
618     dccp_t      *dccp = connp->conn_dccp;
619     dccpha_t     *dccpha;
620     uint32_t      size;
621     uint32_t      offset_size;
622     uint_t        ip_hdr_len = ira->ira_ip_hdr_length;
623
624     cmn_err(CE_NOTE, "dccp_input.c: dccp_check_input");
625
626     size = msgdsize(mp) - (ip_hdr_len);
627     dccpha = (dccpha_t *)&mp->b_rptr[ip_hdr_len];
628     offset_size = dccpha->dha_offset * 4;
629
630     cmn_err(CE_NOTE, "size: %d", size);
631     cmn_err(CE_NOTE, "offset_size: %d", offset_size);
632
633     /* Is packet shorter than 12 bytes? */
634     if (size < DCCP_MIN_HEADER_LENGTH) {
635         cmn_err(CE_NOTE, "header < 12");
636         return (0);
637     }
638
639     /* Data offset is greater than the packet itself */
640     if (offset_size > size) {
641         cmn_err(CE_NOTE, "offset > size");
642         return (0);
643     }
644
645     /* Check if known packet type */
646     if (dccpha->dha_type > DCCP_PKT_SYNCACK) {
647         cmn_err(CE_NOTE, "type unknown");
648         return (0);
649     }
650
651     /*
652      * Check data offset for this packet type and
653      * Data, Ack, or DataAck and P.X == 0
654      */
655     if (dccpha->dha_x == 0) {

```

```

656         switch (dccpha->dha_type) {
657             case DCCP_PKT_DATA:
658                 if (size < 12 || offset_size < 12)
659                     return (0);
660                 break;
661             case DCCP_PKT_ACK:
662             case DCCP_PKT_DATAACK:
663                 if (size < 16 || offset_size < 16)
664                     return (0);
665                 break;
666             default:
667                 return (0);
668         }
669     } else {
670         switch (dccpha->dha_type) {
671             case DCCP_PKT_REQUEST:
672                 if (size < 20 || offset_size < 20)
673                     return (0);
674                 break;
675             case DCCP_PKT_RESPONSE:
676             case DCCP_PKT_RESET:
677                 if (size < 28 || offset_size < 28)
678                     return (0);
679                 break;
680             case DCCP_PKT_DATA:
681                 if (size < 16 || offset_size < 16)
682                     return (0);
683                 break;
684             case DCCP_PKT_ACK:
685             case DCCP_PKT_DATAACK:
686             case DCCP_PKT_CLOSEREQ:
687             case DCCP_PKT_CLOSE:
688             case DCCP_PKT_SYNC:
689             case DCCP_PKT_SYNACK:
690                 if (size < 24 || offset_size < 24)
691                     return (0);
692                 break;
693             default:
694                 return (0);
695         }
696     }
697
698     /* Checksum coverage check */
699     if (dccpha->dha_cscov != 0 && ((dccpha->dha_offset +
700         dccpha->dha_cscov - 1) * 4) > size) {
701         cmn_err(CE_NOTE, "cscov too large");
702         return (0);
703     }
704
705     return (1);
706 }
707
708 /*
709  * After a request-response-ack all packets end up here.
710  */
711 void
712 dccp_input_data(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *ira)
713 {
714     conn_t      *connp = (conn_t *)arg;
715     squeue_t    *sqp = (squeue_t *)arg2;
716     dccp_t      *dccp = connp->conn_dccp;
717     dccp_stack_t *dccps = dccp->dccp_dccps;
718     dccpha_t    *dccpha;
719     dccpha_ack_t *dccpha_ack;
720     dccpha_ext_t *dccpha_ext;
721     mblk_t      *mpl;

```

```

722     uint64_t      seq_seq;
723     uint64_t      seq_ack;
724     uchar_t      *iphdr;
725     uchar_t      *rptr;
726     sock_upcalls_t *sockupcalls;
727     ip_pkt_t      ipp;
728     uint_t        ip_hdr_len;
729     uint_t        len;
730     int           seg_len;
731     int           pkt_len;
732     int           hdr_length;
733     int           error;
734
735     cmn_err(CE_NOTE, "dccp_input.c: dccp_input_data");
736
737     ASSERT(DB_TYPE(mp) == M_DATA);
738     ASSERT(mp->b_next == NULL);
739
740     iphdr = mp->b_rptr;
741     rptr = mp->b_rptr;
742     ASSERT(OK_32PTR(rptr));
743
744     /* Check basic characteristics */
745     if (dccp_check_input(connp, mp, ira) == 0) {
746         cmn_err(CE_NOTE, "rejected packet");
747         return;
748     }
749
750     ip_hdr_len = ira->ira_ip_hdr_length;
751     if (connp->conn_rcv_ancillary.crb_all != 0) {
752         /*
753          * Record packet information in the ip_pkt_t
754          */
755         ipp.ipp_fields = 0;
756         if (ira->ira_flags & IRAF_IS_IPV4) {
757             (void) ip_find_hdr_v4((ipha_t *)rptr, &ipp,
758                 B_FALSE);
759         } else {
760             uint8_t nexthdrp;
761
762             /*
763              * IPv6 packets can only be received by applications
764              * that are prepared to receive IPv6 addresses.
765              * The IP fanout must ensure this.
766              */
767             ASSERT(connp->conn_family == AF_INET6);
768
769             (void) ip_find_hdr_v6(mp, (ip6_t *)rptr, B_TRUE, &ipp,
770                 &nexthdrp);
771             ASSERT(nexthdrp == IPPROTO_DCCP);
772
773             /* Could have caused a pullup? */
774             iphdr = mp->b_rptr;
775             rptr = mp->b_rptr;
776         }
777     }
778
779     len = ip_hdr_len;
780     dccpha = (dccpha_t *)&rptr[len];
781
782     ASSERT((uintptr_t)(mp->b_wptr - rptr) <= (uintptr_t)INT_MAX);
783     seg_len = (int)(mp->b_wptr - rptr) -
784         (ip_hdr_len + DCCP_HDR_LENGTH(dccpha));
785     if ((mpl = mp->b_cont) != NULL && mpl->b_datap->db_type == M_DATA) {
786         do {
787             ASSERT((uintptr_t)(mpl->b_wptr - mpl->b_rptr) <=

```



```

788         (uintptr_t)INT_MAX);
789         seq_len += (int)(mpl->b_wptr - mpl->b_rptr);
790     } while ((mpl = mpl->b_cont) != NULL &&
791             mpl->b_datap->db_type == M_DATA);
792 }

794 DTRACE_DCCP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, connp->conn_ixa,
795             __dtrace_dccp_void_ip_t *, iphdr, dccp_t *, dccp,
796             __dtrace_dccp_dccph_t *, dccpha);

798 if (sqp != NULL) {
799     dccp->dccp_last_rcv_time = LBOLT_FASTPATH;
800 }

802 BUMP_LOCAL(dccp->dccp_ibsegs);
803 DTRACE_PROBE2(dccp__trace_rcv, mblk_t *, mp, dccp_t *, dccp);

805 sockupcalls = connp->conn_upcalls;

807 len += sizeof(dccpha_t);
808 dccpha_ext = (dccpha_ext_t *)&rptr[len];

810 /*
811  * Sequence number.
812  */
813 if (dccpha->dha_x == 1) {
814     seq_seq = ntohs(dccpha->dha_seq);
815     seq_seq = seq_seq << 32;
816     seq_seq |= ntohl(dccpha_ext->dha_ext_seq);
817 } else {
818     /* XXX */
819 }

821 dccp->dccp_gsr = seq_seq;

823 /*
824  * Acknowledgement Number.
825  */
826 if (dccpha->dha_type != DCCP_PKT_DATA) {
827     dccpha_ack = (dccpha_ack_t *)&rptr[ip_hdr_len +
828         sizeof(dccpha_t) + sizeof(dccpha_ext_t)];
829     seq_ack = ntohs(dccpha_ack->dha_ack_high);
830     seq_ack = seq_ack << 32;
831     seq_ack |= ntohl(dccpha_ack->dha_ack_low);

833     dccp->dccp_gar = seq_ack;
834 }

837 switch (dccp->dccp_state) {
838 case DCCPS_REQUEST:
839     cmn_err(CE_NOTE, "DCCPS_REQUEST");
840     break;
841 case DCCPS_RESPOND:
842     cmn_err(CE_NOTE, "DCCPS_RESPOND");
843     break;
844 case DCCPS_PARTOPEN:
845     cmn_err(CE_NOTE, "DCCPS_PARTOPEN");
846     break;
847 case DCCPS_LISTEN:
848     cmn_err(CE_NOTE, "DCCPS_LISTEN");
849     break;
850 case DCCPS_BOUND:
851     cmn_err(CE_NOTE, "DCCPS_BOUND");
852     break;
853 case DCCPS_OPEN:

```

```

854     cmn_err(CE_NOTE, "DCCPS_OPEN");
855     break;
856 default:
857     cmn_err(CE_NOTE, "Unknow state");
858     break;
859 }

861 /*
862  * Check for unexpected packet types.
863  */
864 if ((dccp->dccp_active_open == B_FALSE &&
865     (dccpha->dha_type == DCCP_PKT_RESPONSE ||
866     dccpha->dha_type == DCCP_PKT_CLOSEREQ)) ||
867     (dccp->dccp_state >= DCCPS_OPEN &&
868     (dccpha->dha_type == DCCP_PACKET_REQUEST ||
869     dccpha->dha_type == DCCP_PACKET_RESPONSE) &&
870     seq_seq >= dccp->dccp_osr) ||
871     (dccp->dccp_state == DCCPS_RESPOND &&
872     dccpha->dha_type == DCCP_PACKET_DATA)) {
873     mblk_t *sync_mp;

875     sync_mp = dccp_generate_sync(connp);
876     if (sync_mp != NULL)
877         dccp_send_data(dccp, sync_mp);
878     return;
879 }

881 /*
882  * Process options.
883  */
884 error = dccp_process_options(dccp, dccpha);
885 if (error != 0) {
886     mblk_t *reset_mp;

888     cmn_err(CE_NOTE, "dccp_process_options failed.");

890     reset_mp = dccp_generate_reset(connp);
891     dccp->dccp_state = DCCPS_CLOSED;
892     if (reset_mp != NULL) {
893         ip_xmit_attr_t *ixa = connp->conn_ixa;

895         ixia->ixia_pktlen = msgdsize(reset_mp);
896         (void) conn_ip_output(reset_mp, ixia);
897     }

899     return;
900 }

902 if (dccp->dccp_state == DCCPS_RESPOND) {
903     dccp->dccp_state = DCCPS_OPEN;
904     //dccp->dccp_osr = DCCP_SEQNO_GET(mp);

906     if (dccp->dccp_active_open) {
907         cmn_err(CE_NOTE, "dccp_active_open");
908         if (!dccp_conn_con(dccp, iphdr, mp, NULL, ira)) {
909             cmn_err(CE_NOTE, "dccp_conn_con failed");
910             freemsg(mp);
911             dccp->dccp_state = DCCPS_RESPOND;
912             return;
913         }

915         DTRACE_DCCP5(connect_established, mblk_t *, NULL,
916             ip_xmit_attr_t *, connp->conn_ixa, void_ip_t *,
917             iphdr, dccp_t *, dccp, dccph_t *, dccpha);
918     } else if (IPCL_IS_NONSTR(connp)) {
919         /*

```

```

920      * Passive open socket
921      */
922      cmn_err(CE_NOTE, "nonstr");

924      CONN_INC_REF(connp);

926      if (!dccp_newconn_notify(dccp, ira)) {
927          cmn_err(CE_NOTE, "dccp_newconn_notify failed");
928          dccp->dccp_state = DCCPS_RESPOND;
929          freemsg(mp);

931          CONN_DEC_REF(connp);
932          return;
933      }

935      /*
936       * dccp_newconn_notify() changes conn_upcalls.
937       */
938      if (connp->conn_upcalls != NULL) {
939          sockupcalls = connp->conn_upcalls;
940      }

942      DTRACE_DCCP5(accept_established, mblk_t *, NULL,
943                  ip_xmit_attr_t *, connp->conn_ixa, void_ip_t *,
944                  iphdr, dccp_t *, dccp, dccph_t *, dccpha);
945      } else {
946          cmn_err(CE_NOTE, "str stream");
947          dccp_t *listener = dccp->dccp_listener;

949          ASSERT(mp != NULL);
950          CONN_INC_REF(connp);
951      }
952  }

954  switch (dccpha->dha_type) {
955  case DCCP_PKT_REQUEST:
956      cmn_err(CE_NOTE, "DCCP_REQUEST");
957      break;
958  case DCCP_PKT_RESPONSE:
959      cmn_err(CE_NOTE, "DCCP_RESPONSE");
960      break;
961  case DCCP_PKT_DATA:
962      cmn_err(CE_NOTE, "DCCP_DATA");
963      break;
964  case DCCP_PKT_ACK:
965      cmn_err(CE_NOTE, "DCCP_PKT_ACK");
966      break;
967  case DCCP_PKT_DATAACK:
968      cmn_err(CE_NOTE, "DCCP_DATAACK");
969      break;
970  case DCCP_PKT_CLOSEREQ:
971      cmn_err(CE_NOTE, "DCCP_CLOSEREQ");
972      break;
973  case DCCP_PKT_CLOSE:
974      cmn_err(CE_NOTE, "DCCP_CLOSE");
975      break;
976  case DCCP_PKT_RESET:
977      cmn_err(CE_NOTE, "DCCP_RESET");
978      break;
979  case DCCP_PKT_SYNC:
980      cmn_err(CE_NOTE, "DCCP_SYNC");
981      break;
982  case DCCP_PKT_SYNCACK:
983      cmn_err(CE_NOTE, "DCCP_SYNCACK");
984      break;
985  default:

```

```

986      break;
987  }

989  switch (dccpha->dha_type) {
990  /* case DCCP_PKT_ACK: { */
991  case DCCP_PKT_DATAACK: {
992      mblk_t *ack_mp;

994      ack_mp = dccp_generate_ack(connp);
995      if (ack_mp != NULL)
996          dccp_send_data(dccp, ack_mp);

998      dccp->dccp_state = DCCPS_OPEN;
999      return;
1000  }
1001  case DCCP_PKT_CLOSE: {
1002      mblk_t *reset_mp;

1004      dccp->dccp_reset_code = DCCP_RESET_CLOSED;
1005      dccp->dccp_reset_data[0] = 0;
1006      dccp->dccp_reset_data[1] = 0;
1007      dccp->dccp_reset_data[2] = 0;
1008      reset_mp = dccp_generate_reset(connp);
1009      dccp_send_data(dccp, reset_mp);
1010      dccp->dccp_state = DCCPS_CLOSED;
1011      return;
1012  }
1013  default:
1014      break;
1015  }

1017  hdr_length = ira->ira_ip_hdr_length;
1018  hdr_length += DCCP_HDR_LENGTH(dccpha);
1019  pkt_len = ira->ira_pktlen;

1021  mp->b_wptr = rptr + pkt_len;
1022  mp->b_rptr = (uchar_t *)&mp->b_rptr[hdr_length];
1023  pkt_len -= hdr_length;

1025  if (IPCL_IS_NONSTR(connp)) {
1026      /*
1027       * Non-STREAMS socket.
1028       */
1029      boolean_t push;
1030      int error;

1032      if ((*sockupcalls->su_rcv)(connp->conn_upper_handle,
1033          mp, pkt_len, 0, &error, NULL) <= 0) {
1034          cmn_err(CE_NOTE, "su_rcv failed");
1035          ASSERT(error != EOPNOTSUPP);
1036      }
1037  } else if (dccp->dccp_listener != NULL) {
1038      // dccp_rcv_enqueue(dccp, mp, seq_len, ira->ira_cred);
1039  } else {
1040      /*
1041       * Active-STREAMS socket.
1042       */
1043  }
1044  }
1045  #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp_misc.c

1

```
*****
3084 Sat Aug 18 10:37:10 2012
new/usr/src/uts/common/inet/dccp/dccp_misc.c
dccp: add dccp_misc.c
*****

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 contains various helper functions.
29  */

31 #include <sys/types.h>
32 #include <sys/strlog.h>
33 #include <sys/policy.h>
34 #include <sys/strsun.h>

36 #include <inet/common.h>
37 #include <inet/dccp_impl.h>

39 /*
40  * When a CPU is added, we need to allocate the per CPU stats struct.
41  */
42 void
43 dccp_stack_cpu_add(dccp_stack_t *dccps, processorid_t cpu_seqid)
44 {
45     int    i;

47     if (cpu_seqid < dccps->dccps_sc_cnt) {
48         return;
49     }

51     for (i = dccps->dccps_sc_cnt; i <= cpu_seqid; i++) {
52         ASSERT(dccps->dccps_sc[i] == NULL);
53         dccps->dccps_sc[i] = kmem_zalloc(sizeof (dccp_stats_cpu_t),
54                                         KM_SLEEP);
55     }

57     membar_producer();
58     dccps->dccps_sc_cnt = cpu_seqid + 1;
59 }

61 /*
```

new/usr/src/uts/common/inet/dccp/dccp_misc.c

2

```
62  * Diagnostic routine used to return a string associated with the dccp state.
63  */
64 char *
65 dccp_display(dccp_t *dccp, char *sup_buf, char format)
66 {
67     conn_t      *connp;
68     in6_addr_t   local;
69     in6_addr_t   remote;
70     char         local_addrbuf[INET6_ADDRSTRLEN];
71     char         remote_addrbuf[INET6_ADDRSTRLEN];
72     char         priv_buf[INET6_ADDRSTRLEN * 2 + 80];
73     char         buf1[30];
74     char         *buf;
75     char         *cp;

77     if (sup_buf != NULL) {
78         buf = sup_buf;
79     } else {
80         buf = priv_buf;
81     }

83     if (dccp == NULL) {
84         return ("NULL_DCCP");
85     }

87     connp = dccp->dccp_connp;
88     switch (dccp->dccp_state) {
89     case DCCPS_CLOSED:
90         cp = "DCCP_CLOSED";
91         break;
92     default:
93         (void) mi_sprintf(buf1, "DCCPUnkState(%d)", dccp->dccp_state);
94         cp = buf1;
95         break;
96     }

98     switch (format) {
99     case DISP_ADDR_AND_PORT:
100         if (connp->conn_ipversion == IPV4_VERSION) {
101             IN6_IPADDR_TO_V4MAPPED(connp->conn_laddr_v4, &local);
102             IN6_IPADDR_TO_V4MAPPED(connp->conn_faddr_v4, &remote);
103         } else {
104             local = connp->conn_laddr_v6;
105             remote = connp->conn_faddr_v6;
106         }

108         (void) inet_ntop(AF_INET6, &local, local_addrbuf,
109                         sizeof (local_addrbuf));
110         (void) inet_ntop(AF_INET6, &remote, remote_addrbuf,
111                         sizeof (remote_addrbuf));
112         (void) mi_sprintf(buf, "[%s.%u, %s.%u] %s",
113                         local_addrbuf, ntohs(connp->conn_lport), remote_addrbuf,
114                         ntohs(connp->conn_fport), cp);
115         break;
116     case DISP_PORT_ONLY:
117     default:
118         (void) mi_sprintf(buf, "[%u, %u] %s",
119                         ntohs(connp->conn_lport), ntohs(connp->conn_fport), cp);
120         break;
121     }

123     return (buf);
124 }
125 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_opt_data.c

1

```
*****
7532 Sat Aug 18 10:37:10 2012
new/usr/src/uts/common/inet/dccp/dccp_opt_data.c
dccp: starting with options
*****

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 Hoeppner. 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/dccp_impl.h>
46 #include <inet/ip.h>
47 #include <inet/optcom.h>
48 #include <netinet/ip.h>

50 #include <sys/cmn_err.h>

52 static int dccp_opt_default(queue_t *, int, int, uchar_t *);

54 /*
55  * Supported options.
56 */
57 opdes_t dccp_opt_arr[] = {
58 { SO_DEBUG, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
59 };

61 /*
```

new/usr/src/uts/common/inet/dccp/dccp_opt_data.c

2

```
62 * Supported levels.
63 */
64 optlevel_t dccp_valid_levels_arr[] = {
65     SOL_SOCKET,
66 };

68 #define DCCP_OPT_ARR_CNT      A_CNT(dccp_opt_arr)
69 #define DCCP_VALID_LEVELS_CNT A_CNT(dccp_valid_levels_arr)

71 uint_t dccp_max_optsize;

73 /*
74  * Options database object.
75 */
76 optdb_obj_t dccp_opt_obj = {
77     dccp_opt_default,
78     dccp_tpi_opt_get,
79     dccp_tpi_opt_set,
80     DCCP_OPT_ARR_CNT,
81     dccp_opt_arr,
82     DCCP_VALID_LEVELS_CNT,
83     dccp_valid_levels_arr,
84 };

86 /*
87  * Default value for certain options.
88 */
89 int
90 dccp_opt_default(queue_t *q, int level, int name, uchar_t *ptr)
91 {
92     dccp_stack_t    *dccps = Q_TO_DCCP(q)->dccp_dccps;
93     int32_t          *il = (int32_t *)ptr;

95     return (sizeof (int));
96 }

98 int
99 dccp_opt_get(conn_t *connp, int level, int name, uchar_t *ptr)
100 {
101     dccp_t          *dccp = connp->conn_dccp;
102     conn_opt_arg_t   coas;
103     int              *il = (int *)ptr;
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_DCCP:
116         switch (name) {
117         case DCCP_NOTIFY_THRESHOLD:
118             *il = dccp->dccp_first_timer_threshold;
119             return (sizeof (int));
120         case DCCP_ABORT_THRESHOLD:
121             *il = dccp->dccp_second_timer_threshold;
122             return (sizeof (int));
123         case DCCP_CONN_NOTIFY_THRESHOLD:
124             *il = dccp->dccp_first_ctimer_threshold;
125             return (sizeof (int));
126         case DCCP_CONN_ABORT_THRESHOLD:
127             *il = dccp->dccp_second_ctimer_threshold;
```

```

128         return (sizeof (int));
129     case DCCP_KEEPA_LIVE_THRESHOLD:
130         *il = dccp->dccp_ka_interval;
131         return (sizeof (int));
132     case DCCP_KEEPI_NTVL:
133         *il = dccp->dccp_ka_rinterval / 1000;
134         return (sizeof (int));
135     }
136     break;
137 case IPPROTO_IP:
138     break;
139 case IPPROTO_IPV6:
140     break;
141 }

143 mutex_enter(&connp->conn_lock);
144 retval = conn_opt_get(&coas, level, name, ptr);
145 mutex_exit(&connp->conn_lock);

147 return (retval);
148 }

150 /* ARGSUSED */
151 int
152 dccp_opt_set(conn_t *connp, uint_t optset_context, int level, int name,
153             uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
154             void *thisdg_attrs, cred_t *cr)
155 {
156     dccp_t      *dccp = connp->conn_dccp;
157     dccp_stack_t *dccps = dccp->dccp_dccps;
158     conn_opt_arg_t coas;
159     uint32_t      val = *((uint32_t *)invalp);
160     int           *il = (int *)invalp;
161     boolean_t      onoff = (*il == 0) ? B_FALSE : B_TRUE;
162     boolean_t      checkonly = B_FALSE;
163     int           error;

165     coas.coa_connp = connp;
166     coas.coa_ixa = connp->conn_ixa;
167     coas.coa_ipp = &connp->conn_xmit_ipp;
168     coas.coa_ancillary = B_FALSE;
169     coas.coa_changed = 0;

171     switch (optset_context) {
172     case SETFN_OPTCOM_CHECKONLY:
173         checkonly = B_TRUE;
174         if (inlen == 0) {
175             *outlenp = 0;
176             return (0);
177         }
178         break;
179     case SETFN_OPTCOM_NEGOTIATE:
180         break;
181     case SETFN_UD_NEGOTIATE:
182     case SETFN_CONN_NEGOTIATE:
183         break;
184     default:
185         /*
186          * We should never get here.
187          */
188         *outlenp = 0;
189         return (EINVAL);
190     }

192     ASSERT((optset_context != SETFN_OPTCOM_CHECKONLY) ||
193           (optset_context == SETFN_OPTCOM_CHECKONLY && inlen != 0));

```

```

195     /*
196      * No ancillary data should be sent down.
197      */
198     ASSERT(thisdg_attrs == NULL);

200     switch (level) {
201     case SOL_SOCKET:
202         switch (name) {
203         case SO_KEEPA_LIVE:
204             if (checkonly) {
205                 /* Check only case */
206                 break;
207             }

209             if (!onoff) {
210                 if (connp->conn_keepalive) {
211                     if (dccp->dccp_ka_tid != 0) {
212                         (void) DCCP_TIMER_CANCEL(dccp,
213                                                  dccp->dccp_ka_tid);
214                         dccp->dccp_ka_tid = 0;
215                     }
216                     connp->conn_keepalive = 0;
217                 }
218                 break;
219             }

221             if (!connp->conn_keepalive) {
222                 /* Crank up the keepalive timer */
223                 dccp->dccp_ka_last_intrvl = 0;
224                 dccp->dccp_ka_tid = DCCP_TIMER(dccp,
225                                                  dccp_keepalive_timer, dccp->dccp_ka_interval
226                                                  connp->conn_keepalive = 1;
227             }
228             break;
229         }
230         break;
231     case IPPROTO_DCCP:
232         switch (name) {
233         case DCCP_NOTIFY_THRESHOLD:
234             if (!checkonly)
235                 dccp->dccp_first_timer_threshold = *il;
236             break;
237         case DCCP_ABORT_THRESHOLD:
238             if (!checkonly)
239                 dccp->dccp_second_timer_threshold = *il;
240         case DCCP_CONN_NOTIFY_THRESHOLD:
241             if (!checkonly)
242                 dccp->dccp_first_ctimer_threshold = *il;
243             break;
244         case DCCP_CONN_ABORT_THRESHOLD:
245             if (!checkonly)
246                 dccp->dccp_second_ctimer_threshold = *il;
247             break;
248         case DCCP_KEEPI_DLE:
249             *il *= 1000;
250             /* FALLTHRU */
251         case DCCP_KEEPA_LIVE_THRESHOLD:
252             if (checkonly)
253                 break;

255             if (*il < dccps->dccps_keepalive_interval_low ||
256                 *il > dccps->dccps_keepalive_interval_high) {
257                 *outlenp = 0;
258                 return (EINVAL);
259             }

```

```

261         if (*il != dccp->dccp_ka_interval) {
262             dccp->dccp_ka_interval = *il;

264             if (dccp->dccp_ka_tid != 0) {
265                 ASSERT(connp->conn_keepalive);
266                 (void) DCCP_TIMER_CANCEL(dccp,
267                     dccp->dccp_ka_tid);
268                 dccp->dccp_ka_last_intrvl = 0;
269                 dccp->dccp_ka_tid = DCCP_TIMER(dccp,
270                     dccp_keepalive_timer,
271                     dccp->dccp_ka_interval);
272             }
273         }
274         break;
275     }
276     break;
277 case IPPROTO_IP:
278     break;
279 case IPPROTO_IPV6:
280     break;
281 }

283 error = conn_opt_set(&coas, level, name, inlen, invalp,
284     B_FALSE, cr);
285 if (error != 0) {
286     *outlenp = 0;
287     return (error);
288 }

290 /*
291  * Common case of OK return with outval same as inval.
292  */
293 if (invalp != outvalp) {
294     (void) bcopy(invalp, outvalp, inlen);
295 }
296 *outlenp = inlen;

298 if (coas.coa_changed && COA_HEADER_CHANGED) {
299     /* If we are connected we rebuilt the headers */
300     if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
301         !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
302         error = dccp_build_hdrs(dccp);
303         if (error != 0) {
304             return (error);
305         }
306     }
307 }

309 if (coas.coa_changed & COA_ROUTE_CHANGED) {
310     in6_addr_t     nexthop;

312     ip_attr_nexthop(&connp->conn_xmit_ipp, connp->conn_ixa,
313         &connp->conn_faddr_v6, &nexthop);

315     if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
316         !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
317         (void) ip_attr_connect(connp, connp->conn_ixa,
318             &connp->conn_laddr_v6, &connp->conn_faddr_v6,
319             &nexthop, connp->conn_fport, NULL, NULL,
320             IPDF_VERIFY_DST);
321     }
322 }

324 /* XXX */

```

```

326         return (0);
327     }
328 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp_options.c

1

6895 Sat Aug 18 10:37:10 2012

new/usr/src/uts/common/inet/dccp/dccp_options.c

dccp: split options and feature functions

```
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 Hoeppner. All rights reserved.
14 */

16 #include <sys/types.h>
17 #include <sys/stream.h>
18 #include <sys/debug.h>
19 #include <sys/cmn_err.h>

21 #include <inet/dccp_impl.h>
22 #include <inet/dccp_stack.h>

24 /*
25  * This file contains functions to parse and process DCCP options.
26 */

29 /*
30  * Parse the options in a DCCP header.
31 */
32 int
33 dccp_parse_options(dccp_t *dccp, dccpha_t *dccpha)
34 {
35     uchar_t      *end;
36     uchar_t      *up;
37     uint8_t       dccp_type;
38     uint32_t      option_value;
39     uint8_t       option_type;
40     uint8_t       option_length;
41     int           len;
42     int           i;
43     uchar_t       *value;
44     boolean_t     mandatory = B_FALSE;
45     int           error;

47     cmn_err(CE_NOTE, "dccp_features.c: dccp_parse_options");

49     dccp_type = dccpha->dha_type;

51     up = (uchar_t *)dccpha;
52     end = up + DCCP_HDR_LENGTH(dccpha);
53     up += 20;

55     while (up != end) {
56         option_length = 0;
57         option_type = *up++;

59         if (option_type > 31) {
60             if (up == end) {
61                 goto length_error;
```

new/usr/src/uts/common/inet/dccp/dccp_options.c

2

```
62     }

64     option_length = *up++;
65     if (option_length < 2) {
66         goto length_error;
67     }

69     option_length -= 2;
70     value = up;

72     up += option_length;

74     /* Ignore options with greater length then header */
75     if (up > end) {
76         goto length_error;
77     }
78 } else {
79     value = up;
80 }

82 /* Single byte unknow option */
83 if (option_type >= 3 && option_type <=31) {
84     if (mandatory) {
85         error = DCCP_RESET_MANDATORY_ERROR;
86     } else {
87         error = DCCP_RESET_OPTION_ERROR;
88     }

90     dccp->dccp_reset_code = error;
91     /*
92      * this should be correct:
93      * dccp->dccp_reset_data[0] = option_type;
94      * dccp->dccp_reset_data[1] = value[0];
95      *
96      * error in the test suit?
97      */
98     dccp->dccp_reset_data[0] = value[0];
99     dccp->dccp_reset_data[1] = 0;
100    dccp->dccp_reset_data[2] = 0;

102    return (-1);
103 }

105 if (option_type >= 45 && option_type <=127)
106     break;

109 switch (option_type) {
110 case DCCP_OPTION_PADDING:
111     cmn_err(CE_NOTE, "PADDING");
112     break;
113 case DCCP_OPTION_MANDATORY:
114     cmn_err(CE_NOTE, "MANDATORY");
115     if (mandatory)
116         goto option_error;

118     if (dccp_type != DCCP_PKT_DATA)
119         mandatory = B_TRUE;
120     break;
121 case DCCP_OPTION_SLOW_RECEIVER:
122     cmn_err(CE_NOTE, "SLOW RECEIVER");
123     break;
124 case DCCP_OPTION_CHANGE_L:
125 case DCCP_OPTION_CONFIRM_L:
126 case DCCP_OPTION_CHANGE_R:
127 case DCCP_OPTION_CONFIRM_R:
```

```

128         if (dccp_type == DCCP_PACKET_DATA)
129             break;

131         if (option_length == 0)
132             goto option_error;

134         error = dccp_parse_feature(dccp, option_type, option_len
135             value, mandatory);
136         if (error != 0)
137             goto feature_error;

139         break;
140     case DCCP_OPTION_INIT_COOKIE:
141         cmn_err(CE_NOTE, "INIT COOKIE");
142         break;
143     case DCCP_OPTION_NDP_COUNT:
144         cmn_err(CE_NOTE, "NDP COUNT");
145         if (option_length > 6)
146             goto option_error;
147         break;
148     case DCCP_OPTION_ACK_VECTOR_1:
149         cmn_err(CE_NOTE, "ACK VECTOR 1");
150         break;
151     case DCCP_OPTION_ACK_VECTOR_2:
152         cmn_err(CE_NOTE, "ACK VECTOR 2");
153         break;
154     case DCCP_OPTION_DATA_DROPPED:
155         cmn_err(CE_NOTE, "DATA DROPPED");
156         break;
157     case DCCP_OPTION_TIMESTAMP:
158         cmn_err(CE_NOTE, "TIMESTAMP");
159         if (option_length != 4)
160             goto option_error;

162         option_value = BE32_TO_U32(value);
163         if (option_value == 0) {
164             cmn_err(CE_NOTE, "Zero timestamp");
165             break;
166         }

168         dccp->dccp_timestamp_echo = ntohs(option_value);
169         dccp->dccp_timestamp = TICK_TO_MSEC(LBOLT_FASTPATH);
170         break;
171     case DCCP_OPTION_TIMESTAMP_ECHO:
172         cmn_err(CE_NOTE, "TIMESTAMP ECHO");
173         if (option_length != 4 && option_length != 6 &&
174             option_length != 8) {
175             goto option_error;
176         }

178         option_value = BE32_TO_U32(value);

180         option_length -= 4;
181         break;
182     case DCCP_OPTION_ELAPSED_TIME:
183         cmn_err(CE_NOTE, "ELAPSES TIME");
184         switch (option_length) {
185             case 2:
186                 break;
187             case 4:
188                 break;
189             default:
190                 goto option_error;
191         }
192         break;
193     case DCCP_OPTION_DATA_CHECKSUM:

```

```

194         cmn_err(CE_NOTE, "DATA CHECKSUM");
195         break;

197         default:
198             cmn_err(CE_NOTE, "unknow option");
199             break;
200     }

202     if (option_type != DCCP_OPTION_MANDATORY) {
203         mandatory = B_FALSE;
204     }
205 }

207     if (mandatory)
208         goto option_error;

210 length_error:
211     return (0);

213 option_error:
214     if (mandatory) {
215         error = DCCP_RESET_MANDATORY_ERROR;
216     } else {
217         error = DCCP_RESET_OPTION_ERROR;
218     }

220     cmn_err(CE_NOTE, "setting error code");

222 feature_error:
223     dccp->dccp_reset_code = error;
224     dccp->dccp_reset_data[0] = option_type;
225     dccp->dccp_reset_data[1] = option_length > 0 ? value[0] : 0;
226     dccp->dccp_reset_data[2] = option_length > 1 ? value[1] : 0;

228     return (-1);
229 }

231 int
232 dccp_process_options(dccp_t *dccp, dccpha_t *dccpha)
233 {
234     int    error;

236     cmn_err(CE_NOTE, "dccp_features.c: dccp_process_features");

238     error = dccp_parse_options(dccp, dccpha);

240     return (error);
241 }

243 int
244 dccp_generate_options(dccp_t *dccp, void **opt, size_t *opt_len)
245 {
246     dccp_feature_t    *feature = NULL;
247     uint8_t           buf[1024]; /* XXX */
248     uint8_t           option_type;
249     uint_t             len = 0;
250     uint_t             total_len = 0;
251     void               *options = NULL;
252     int                rest;

254     cmn_err(CE_NOTE, "dccp_features.c: dccp_generate_options");

256     for (feature = list_head(&dccp->dccp_features); feature;
257         feature = list_next(&dccp->dccp_features, feature)) {
258         if (feature->df_option == DCCP_OPTION_CHANGE_L) {
259             option_type = DCCP_OPTION_CONFIRM_R;

```



```

260     } else {
261         option_type = DCCP_OPTION_CONFIRM_L;
262     }
263 /*
264     if (feature->df_mandatory == B_TRUE) {
265         buf[len] = DCCP_OPTION_MANDATORY;
266         len++;
267     }
268 */
269     if (feature->df_type == DCCP_FEATURE_CCID) {
270         cmn_err(CE_NOTE, "FOUND DCCP_FEATURE_CCID");
271
272         buf[len] = option_type;
273         len++;
274         buf[len] = 4;
275         len++;
276         buf[len] = DCCP_FEATURE_CCID;
277         len++;
278         buf[len] = 2;
279         len++;
280     }
281
282     if (feature->df_type == DCCP_FEATURE_ALLOW_SHORT_SEQNOS) {
283         buf[len] = option_type;
284         len++;
285         buf[len] = 4;
286         len++;
287         buf[len] = feature->df_type;
288         len++;
289         buf[len] = 0;
290         len++;
291     }
292
293     if (feature->df_type == DCCP_FEATURE_ECN_INCAPABLE) {
294         buf[len] = option_type;
295         len++;
296         buf[len] = 4;
297         len++;
298         buf[len] = feature->df_type;
299         len++;
300         buf[len] = 1;
301         len++;
302     }
303 }
304
305 if (dccp->dccp_timestamp_echo != 0) {
306     uint32_t    elapsed;
307     int         elapsed_length;
308     clock_t     now;
309
310     buf[len] = DCCP_OPTION_TIMESTAMP_ECHO;
311     len++;
312     buf[len] = 10;
313     len++;
314
315     now = TICK_TO_MSEC(LBOLT_FASTPATH);
316     elapsed = now - dccp->dccp_timestamp;
317
318     dccp->dccp_timestamp_echo = 0;
319 }
320
321 if (len != 0) {
322     total_len = ((len + (4 - 1)) / 4) * 4;
323     options = kmem_zalloc(total_len, KM_SLEEP);
324     if (options == NULL) {
325

```

```

326         cmn_err(CE_NOTE, "kmem_zalloc failed");
327         return (ENOMEM);
328     }
329     memcpy(options, buf, len);
330 }
331
332 *opt = options;
333 *opt_len = len;
334
335 return (0);
336 }
337 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp_output.c

1

25748 Sat Aug 18 10:37:10 2012

new/usr/src/uts/common/inet/dccp/dccp_output.c

dccp: clean up

```
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 Hoeppner. All rights reserved.
29 */
30
31 /*
32  * Functions related to the output path.
33 */
34
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/squeue_impl.h>
46 #include <sys/squeue.h>
47 #include <sys/tsol/tnet.h>
48
49 #include <inet/common.h>
50 #include <inet/dccp_impl.h>
51 #include <inet/dccp_stack.h>
52 #include <inet/ip.h>
53 #include <inet/ipsec_impl.h>
54
55 #include <sys/cmn_err.h>
56
57 static void dccp_xmit_early_reset(char *, mblk_t *, uint32_t, uint32_t,
58 int, ip_rcv_attr_t *, ip_stack_t *, conn_t *);
59 static boolean_t dccp_send_rst_chk(dccp_stack_t *);
60 static int dccp_xmit_end(dccp_t *);
61 static mblk_t *dccp_generate_common(conn_t *, uint8_t, uint_t);
```

new/usr/src/uts/common/inet/dccp/dccp_output.c

2

```
63 /*
64  * STREAMS
65 */
66 void
67 dccp_wput(queue_t *q, mblk_t *mp)
68 {
69     cmn_err(CE_NOTE, "dccp_output.c: dccp_wput");
70 }
71
72 /*
73  * Fast path write put.
74 */
75 void
76 dccp_wput_data(dccp_t *dccp, mblk_t *mp, boolean_t urgent)
77 {
78     cmn_err(CE_NOTE, "dccp_output.c: dccp_wput_data");
79 }
80
81 /*
82  *
83 */
84 void
85 dccp_wput_sock(queue_t *wq, mblk_t *mp)
86 {
87     conn_t *connp = Q_TO_CONN(wq);
88     dccp_t *dccp = connp->conn_dccp;
89     struct T_capability_req *car = (struct T_capability_req *)mp->b_rptr;
90
91     cmn_err(CE_NOTE, "dccp_wput_sock");
92
93     ASSERT(wq->q_qinfo == &dccp_sock_winit);
94     wq->q_qinfo = &dccp_winit;
95
96     ASSERT(IPCL_IS_TCP(connp));
97     ASSERT(DCCP_IS_SOCKET(dccp));
98
99     if (DB_TYPE(mp) == M_PCPROTO &&
100         MBLKL(mp) == sizeof (struct T_capability_req) &&
101         car->PRIM_type == T_CAPABILITY_REQ) {
102         dccp_capability_req(dccp, mp);
103         return;
104     }
105
106     dccp_wput(wq, mp);
107 }
108
109 /* ARGSUSED */
110 void
111 dccp_wput_fallback(queue_t *eq, mblk_t *mp)
112 {
113     cmn_err(CE_NOTE, "dccp_output.c: dccp_wput_fallback");
114
115     #ifdef DEBUG
116     cmn_err(CE_CONT, "tcp_wput_fallback: Message during fallback \n");
117     #endif /* DEBUG */
118
119     freemsg(mp);
120 }
121
122 /*
123  * Initiate closedown sequence on an active connection.
124 */
125 static int
126 dccp_xmit_end(dccp_t *dccp)
127 {
```

```

128     conn_t  *connp = dccp->dccp_conn;
129     mblk_t  *mp;

131     /* XXX */

133     return (0);
134 }

136 /*
137  * Output fast path.
138  */
139 void
140 dccp_output(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
141 {
142     conn_t      *connp = (conn_t *)arg;
143     dccp_t      *dccp = connp->conn_dccp;
144     dccp_stack_t *dccps = dccp->dccp_dccps;
145     dccpha_t    *dccpha;
146     mblk_t      *mpl;
147     ip_xmit_attr_t *ixa;
148     struct datab *db;
149     uchar_t     *rptr;
150     uint32_t    msize;
151     uint32_t    sum;
152     int         len;
153     int         plen;

155     cmn_err(CE_NOTE, "dccp_output.c: dccp_output");

157     ASSERT((connp->conn_fanout != NULL && connp->conn_ref >= 4) ||
158         (connp->conn_fanout == NULL && connp->conn_ref >= 3));

160     ASSERT(DB_TYPE(mp) == M_DATA);
161     msize = (mp->b_cont == NULL) ? MBLKL(mp) : msgdsz(mp);

163     ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <= (uintptr_t)INT_MAX);
164     len = (int)(mp->b_wptr - mp->b_rptr);

166     if ((mp->b_cont != NULL) ||
167         (dccp->dccp_state != DCCPS_OPEN) ||
168         (len == 0)) {
169         dccp_wput_data(dccp, mp, B_FALSE);
170         return;
171     }

173     mpl = dupb(mp);
174     if (mpl == NULL) {
175         goto no_memory;
176     }

178     /* Adjust header information */
179     dccpha = dccp->dccp_dccpha;

181     sum = len + connp->conn_ht_ulp_len + connp->conn_sum;
182     sum = (sum >> 16) + (sum & 0xffff);
183     dccpha->dha_sum = htons(sum);

185     DCCPS_BUMP_MIB(dccps, dccpOutDataSegs);
186     DCCPS_UPDATE_MIB(dccps, dccpOutDataBytes, len);
187     BUMP_LOCAL(dccp->dccp_obses);

189     plen = len + connp->conn_ht_iphc_len;

191     ixa = connp->conn_ixa;
192     ixa->ixa_pktlen = plen;

```

```

194     if (ixa->ixa_flags & IXAF_IS_IPV4) {
195         dccp->dccp_ipha->ipha_length = htons(plen);
196     } else {
197         dccp->dccp_ip6h->ip6_plen = htons(plen - IPV6_HDR_LEN);
198     }

200     rptr = mpl->b_rptr;
201     bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);

203     dccp_send_data(dccp, mpl);

205     return;

207 no_memory:
208     return;
209 }

211 void
212 dccp_output_urgent(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
213 {
214     cmn_err(CE_NOTE, "dccp_output.c: dccp_output_urgent");
215 }

217 void
218 dccp_close_output(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
219 {
220     conn_t      *connp = (conn_t *)arg;
221     dccp_t      *dccp = connp->conn_dccp;
222     dccp_stack_t *dccps = dccp->dccp_dccps;
223     char         *msg;

225     cmn_err(CE_NOTE, "dccp_output.c: dccp_close_output");

227     /*
228      * When a non-STREAMS socket is being closed, it does not always
229      * stick around waiting for tcp_close_output to run and can therefore
230      * have dropped a reference already. So adjust the asserts accordingly.
231      */
232     ASSERT((connp->conn_fanout != NULL &&
233         connp->conn_ref >= (IPCL_IS_NONSTR(connp) ? 3 : 4)) ||
234         (connp->conn_fanout == NULL &&
235         connp->conn_ref >= (IPCL_IS_NONSTR(connp) ? 2 : 3)));

237     msg = NULL;
238     switch (dccp->dccp_state) {
239     case DCCPS_CLOSED:
240         break;
241     case DCCPS_BOUND:
242         break;
243     case DCCPS_REQUEST:
244         msg = "dccp_close, during connect";
245         break;
246     case DCCPS_RESPOND:
247         /* FALLTHRU */
248     default:
249         /*
250          * If SO_LINGER has set a zero linger time, abort the
251          * connection with a reset.
252          */
253         if (connp->conn_linger && connp->conn_lingertime == 0) {
254             msg = "dccp_close, zero lingertime";
255             break;
256         }
257     }
258 }

```

```

260 /* ARGSUSED */
261 void
262 dccp_shutdown_output(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
263 {
264     conn_t *connp = (conn_t *)arg;
265     dccp_t *dccp = connp->conn_dccp;
266
267     freemsg(mp);
268
269     if (dccp_xmit_end(dccp) != 0) {
270         /*
271          * We were crossing FINs and got a reset from
272          * the other side. Just ignore it.
273          */
274         if (connp->conn_debug) {
275             (void) strlog(DCCP_MOD_ID, 0, 1,
276                 SL_ERROR|SL_TRACE,
277                 "dccp_shutdown_output() out of state %s",
278                 dccp_display(dccp, NULL, DISP_ADDR_AND_PORT));
279         }
280     }
281 }
282
283 #pragma inline(dccp_send_data)
284
285 void
286 dccp_send_data(dccp_t *dccp, mblk_t *mp)
287 {
288     conn_t *connp = dccp->dccp_conn;
289     int error;
290
291     cmn_err(CE_NOTE, "dccp_output.c: dccp_send_data");
292
293     /* XXX zcopy aware */
294
295     DTRACE_DCCP5(send, mblk_t *, NULL, ip_xmit_attr_t *, connp->conn_ixa,
296         __dtrace_dccp_void_ip_t *, mp->b_rptr, dccp_t *, dccp,
297         __dtrace_dccp_dccph_t *,
298         &mp->b_rptr[connp->conn_ixa->ixa_ip_hdr_length]);
299
300     ASSERT(connp->conn_ixa->ixa_notify_cookie == connp->conn_tcp);
301     error = conn_ip_output(mp, connp->conn_ixa);
302     if (error != 0) {
303         cmn_err(CE_NOTE, "conn_ip_output failed with code %d\n", error);
304     }
305 }
306
307 /*
308  * Send a reset as response to an incoming packet or
309  * reset a connection.
310  */
311 void
312 dccp_xmit_listeners_reset(mblk_t *mp, ip_rcv_attr_t *ira, ip_stack_t *ipst,
313     conn_t *connp)
314 {
315     netstack_t *ns = ipst->ips_netstack;
316     dccp_stack_t *dccps = ns->netstack_dccp;
317     ipsec_stack_t *ipss = dccps->dccps_netstack->netstack_ipsec;
318     dccpha_t *dccpha;
319     ipha_t *ipha;
320     ip6_t *ip6h;
321     uchar_t *rp_ptr;
322     uint32_t seq_len;
323     uint_t ip_hdr_len = ira->ira_ip_hdr_length;
324     boolean_t policy_present;

```

```

326     cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_listeners_reset");
327
328     DCCP_STAT(dccps, dccp_no_listener);
329
330     if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
331         policy_present = ipss->ipsec_inbound_v4_policy_present;
332         ipha = (ipha_t *)mp->b_rptr;
333         ip6h = NULL;
334     } else {
335         policy_present = ipss->ipsec_inbound_v6_policy_present;
336         ipha = NULL;
337         ip6h = (ip6_t *)mp->b_rptr;
338     }
339
340     if (policy_present) {
341         mp = ipsec_check_global_policy(mp, (conn_t *)NULL, ipha, ip6h,
342             ira, ns);
343         if (mp == NULL) {
344             return;
345         }
346     }
347
348     rp_ptr = mp->b_rptr;
349
350     dccpha = (dccpha_t *)&rp_ptr[ip_hdr_len];
351
352     seq_len = msgdsize(mp) - (ip_hdr_len);
353
354     dccp_xmit_early_reset("no dccp, reset", mp, 0,
355         0, 0, ira, ipst, connp);
356 }
357
358 /*
359  * RFC 4340, Section 8.1.3
360  */
361 static void
362 dccp_xmit_early_reset(char *str, mblk_t *mp, uint32_t seq, uint32_t ack, int ctl,
363     ip_rcv_attr_t *ira, ip_stack_t *ipst, conn_t *connp)
364 {
365     dccpha_t *dccpha;
366     dccpha_t *nmp_dccpha;
367     dccpha_ack_t *nmp_dccpha_ack;
368     dccpha_reset_t *dccpha_reset;
369     dccpha_reset_t *nmp_dccpha_reset;
370     dccpha_ext_t *dccpha_ext;
371     dccpha_ext_t *nmp_dccpha_ext;
372     netstack_t *ns = ipst->ips_netstack;
373     dccp_stack_t *dccps = ns->netstack_dccp;
374     ip6_t *ip6h;
375     ipha_t *ipha;
376     ipha_t *nmp_ipha;
377     ip_xmit_attr_t ixas;
378     ip_xmit_attr_t *ixa;
379     in6_addr_t v6addr;
380     ipaddr_t v4addr;
381     mblk_t *nmp;
382     uint64_t pkt_ack;
383     uint_t ip_hdr_len = ira->ira_ip_hdr_length;
384     ushort_t port;
385     ushort_t len;
386
387     cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_early_reset");
388
389     if (!dccp_send_rst_chk(dccps)) {
390         cmn_err(CE_NOTE, "dccp_output.c: not sending reset packet");
391         DCCP_STAT(dccps, dccp_rst_unsent);

```

```

392     freemsg(mp);
393     return;
394 }

396     bzero(&ixas, sizeof (ixas));
397     ixa = &ixas;

399     ixa->ixa_flags |= IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE;
400     ixa->ixa_protocol = IPPROTO_DCCP;
401     ixa->ixa_zoneid = ira->ira_zoneid;
402     ixa->ixa_ifindex = 0;
403     ixa->ixa_ipst = ipst;
404     ixa->ixa_cred = kcred;
405     ixa->ixa_cpuid = NOPID;

407     if (str && dccps->dccps_dbg) {
408         (void) strlog(DCCP_MOD_ID, 0, 1, SL_TRACE,
409             "dccp_xmit_early_reset: '%s', seq 0x%x, ack 0x%x, "
410             "flags 0x%x",
411             str, seq, ack, ctl);
412     }

414     if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
415         ipha = (ipha_t *)mp->b_rptr;
416     } else {
417         /* XXX */
418     }

419     /*
420      * Allocate a new DCCP reset message
421      */
422     len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t) + sizeof (d
423     nmp = allocb(len, BPRI_MED);
424     if (nmp == NULL) {
425         cmn_err(CE_NOTE, "alloc failed");
426         return;
427     }
428     bcopy(mp->b_rptr, nmp->b_wptr, ip_hdr_len + sizeof (dccpha_t));

432     nmp_dccpha = (dccpha_t *)&nmp->b_rptr[ip_hdr_len];
433     nmp_dccpha->dha_offset = 7;

435     if (IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION) {
436         nmp_ipha = (ipha_t *)nmp->b_rptr;

438         nmp_ipha->ipha_length = htons(len);
439         nmp_ipha->ipha_src = ipha->ipha_src;
440         nmp_ipha->ipha_dst = ipha->ipha_dst;

442         ixa->ixa_flags |= IXAF_IS_IPV4;
443         ixa->ixa_ip_hdr_length = ip_hdr_len;
444     } else {
445         cmn_err(CE_NOTE, "not v4");
446     }

448     dccpha = (dccpha_t *)&nmp->b_rptr[ip_hdr_len];

450     nmp->b_wptr = &nmp->b_rptr[len];

452     ixa->ixa_pktlen = len; // ?

454     nmp_dccpha->dha_fport = dccpha->dha_lport;
455     nmp_dccpha->dha_lport = dccpha->dha_fport;
456     nmp_dccpha->dha_type = DCCP_PKT_RESET;
457     nmp_dccpha->dha_x = 1;

```

```

458     nmp_dccpha->dha_res_seq = 0;
459     nmp_dccpha->dha_seq = 0;

461     nmp_dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) +
462         sizeof (dccpha_ack_t) + sizeof (dccpha_reset_t));

464     dccpha_ext = (dccpha_ext_t *)&nmp->b_rptr[ip_hdr_len + sizeof (dccpha_t)]
465     nmp_dccpha_ext = (dccpha_ext_t *)&nmp->b_rptr[ip_hdr_len + sizeof (dccph
466     nmp_dccpha_ext->dha_ext_seq = 0;

468     len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t);
469     nmp_dccpha_ack = (dccpha_ack_t *)&nmp->b_rptr[len];
470     nmp_dccpha_ack->dha_ack_high = dccpha->dha_seq;
471     nmp_dccpha_ack->dha_ack_low = dccpha_ext->dha_ext_seq;
472     nmp_dccpha_ack->dha_ack_reserved = 0;

474     len = ip_hdr_len + sizeof (dccpha_t) + sizeof (dccpha_ext_t) + sizeof (d
475     nmp_dccpha_reset = (dccpha_reset_t *)&nmp->b_rptr[len];

477     if (connp != NULL) {
478         nmp_dccpha_reset->dha_reset_code =
479             connp->conn_dccp->dccp_reset_code;
480     } else {
481         nmp_dccpha_reset->dha_reset_code =
482             DCCP_RESET_CONNECTION_REFUSED;
483     }
484     nmp_dccpha_reset->dha_reset_data[0] = 0;
485     nmp_dccpha_reset->dha_reset_data[1] = 0;
486     nmp_dccpha_reset->dha_reset_data[2] = 0;

488     (void) ip_output_simple(nmp, ixa);

490     ixa_cleanup(ixa);
491 }

493 /*
494  *
495  */
496 static boolean_t
497 dccp_send_rst_chk(dccp_stack_t *dccps)
498 {
499     int64_t now;

501     if (dccps->dccps_rst_sent_rate_enabled != 0) {
502         now = ddi_get_lbolt64();
503         if (TICK_TO_MSEC(now - dccps->dccps_last_rst_intrvl) >
504             1 * SECONDS) {
505             dccps->dccps_last_rst_intrvl = now;
506             dccps->dccps_rst_cnt = 1;
507         } else if (++dccps->dccps_rst_cnt > dccps->dccps_rst_sent_rate)
508             return (B_FALSE);
509     }
510 }

512     return (B_TRUE);
513 }

515 /* ARGSUSED2 */
516 void
517 dccp_send_synack(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *dummy)
518 {
519     conn_t         *econnp = (conn_t *)arg;
520     dccp_t         *dccp = econnp->conn_dccp;
521     ip_xmit_attr_t *ixa = econnp->conn_ixa;

523     cmn_err(CE_NOTE, "dccp_output.c: dccp_send_synack");

```

```

525     /*
526     * Guard against a RESET having blown it away while on the
527     * queue.
528     */
529     if (dccp->dccp_state == DCCPS_CLOSED) {
530         freemsg(mp);
531         return;
532     }

534     ixa->ixa_pktlen = msgdsize(mp);
535     (void) conn_ip_output(mp, ixa);
536 }

538 mblk_t *
539 dccp_xmit_mp(dccp_t *dccp, mblk_t *mp, int32_t max_to_send, int32_t *offset,
540             mblk_t **end_mp, uint32_t seq, boolean_t sendall, uint32_t *seg_len,
541             boolean_t rexmit)
542 {
543     conn_t      *connp = dccp->dccp_connnp;
544     dccp_stack_t *dccps = dccp->dccp_dccps;
545     dccpha_t     *dccpha;
546     dccpha_ext_t *dccpha_ext;
547     dccpha_ack_t *dccpha_ack;
548     dccpha_srv_t *dccpha_srv;
549     ip_xmit_attr_t *ixa = connp->conn_ixa;
550     mblk_t       *mpl;
551     uchar_t      *rptr;
552     ushort_t     len;
553     int           data_length;

555     cmn_err(CE_NOTE, "dccp_output.c: dccp_xmit_mp");

557     // dccpha_t already in iphc_len?
558     len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) + sizeof (dccpha_a

560     mpl = allocb(len, BPRI_MED);
561     if (mpl == NULL) {
562         cmn_err(CE_NOTE, "allocb failed");
563         return (NULL);
564     }

566     data_length = 0;

568     rptr = mpl->b_rptr;
569     mpl->b_wptr = &mpl->b_rptr[len];
570     bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);
571     dccpha = (dccpha_t *) &rptr[ixa->ixa_ip_hdr_length];
572     dccpha->dha_type = DCCP_PKT_RESPONSE;
573     dccpha->dha_offset = 8;
574     dccpha->dha_x = 1;
575     dccpha->dha_ccval = 0;
576     dccpha->dha_cscov = 0;
577     dccpha->dha_reserved = 0;
578     dccpha->dha_res_seq = 0;
579     dccpha->dha_seq = 0;

581     dccpha_ext = (dccpha_ext_t *) &rptr[ixa->ixa_ip_hdr_length + sizeof (dccp
582     dccpha_ext->dha_ext_seq = 0;

584     dccpha_ack = (dccpha_ack_t *) &rptr[ixa->ixa_ip_hdr_length + sizeof (dccp
585     dccpha_ack->dha_ack_reserved = 0;
586     dccpha_ack->dha_ack_high = 0;
587     dccpha_ack->dha_ack_low = 0;

589     dccpha_srv = (dccpha_srv_t *) &rptr[ixa->ixa_ip_hdr_length + sizeof (dccp

```

```

590     dccpha_srv->dha_srv_code = 0;

592     return (mpl);
593 }

595 /*
596 * Generate a DCCP-Response packet.
597 */
598 mblk_t *
599 dccp_generate_response(conn_t *connp, mblk_t *mp)
600 {
601     dccpha_t     *dccpha;
602     dccpha_ext_t *dccpha_ext;
603     dccpha_ack_t *dccpha_ack;
604     dccpha_srv_t *dccpha_srv;
605     mblk_t       *mpl;
606     uint64_t     seq;
607     uint64_t     ack;
608     uint16_t     ack_high;
609     uint32_t     ack_low;
610     //
611     ip_hdr_len = ira->ira_ip_hdr_length;
612     ip_xmit_attr_t *ixa = connp->conn_ixa;
613     uint_t       ip_hdr_len;
614     uint_t       len;
615     uint_t       total_hdr_len;
616     uchar_t      *rptr;
617     dccp_t       *dccp = connp->conn_dccp;
618     void          *options;
619     size_t       opt_len;
620     int           error;

621     cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_packet");

623     ip_hdr_len = ixa->ixa_ip_hdr_length;

625     if (mp == NULL) {
626         cmn_err(CE_NOTE, "NULL pointer mp");
627         return (NULL);
628     }

630     dccpha = (dccpha_t *) &mp->b_rptr[ip_hdr_len];
631     dccpha_ext = (dccpha_ext_t *) &mp->b_rptr[ip_hdr_len + sizeof (dccpha_t)]

633     ack_high = dccpha->dha_seq;
634     ack_low = dccpha_ext->dha_ext_seq;

636     seq = ntohs(ack_high) << 31;
637     seq |= ntohl(ack_low);

639     dccp->dccp_isr = seq;
640     dccp->dccp_gsr = seq;
641     dccp->dccp_swl = seq;
642     dccp->dccp_swh = seq;
643     dccp->dccp_gss++;

645     error = dccp_generate_options(dccp, &options, &opt_len);
646     if (error != 0) {
647         cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_options failed");
648     }
649     cmn_err(CE_NOTE, "generated options len: %d", (int) opt_len);

652     /*
653     * conn_ht_iphc_len = ip_hdr_length (20) + ulp_hdr_length
654     * (20) simple ip header (without vtag or options)
655     */

```

```

656     total_hdr_len = len + connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
657     mpls = allocb(len, BPRI_MED);
658     if (mpls == NULL) {
659         cmn_err(CE_NOTE, "allocb failed");
660         return (NULL);
661     }

663     rptra = mpls->b_rptr;
664     mpls->b_wptr = &mpls->b_rptr[len];

666     bcopy(options, &mpls->b_rptr[len-opt_len], opt_len);
667     bcopy(connp->conn_ht_iphc, rptra, connp->conn_ht_iphc_len);
668     dccpha = (dccpha_t *) &rptra[ip_hdr_len];

670     dccpha->dha_type = DCCP_PKT_RESPONSE;
671     dccpha->dha_offset = 7 + (opt_len / 4);
672     dccpha->dha_x = 1;
673     dccpha->dha_ccval = 0;
674     dccpha->dha_cscov = 0;
675     dccpha->dha_reserved = 0;
676     dccpha->dha_res_seq = 0;
677     dccpha->dha_seq = htons(dccp->dccp_gss >> 32);
678     dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) + size

681     dccpha_ext = (dccpha_ext_t *) &mpls->b_rptr[ip_hdr_len + sizeof (dccpha_t)
682     dccpha_ext->dha_ext_seq = htonl(dccp->dccp_gss & 0xffffffff);

684     dccpha_ack = (dccpha_ack_t *) &mpls->b_rptr[ip_hdr_len + sizeof (dccpha_t)
685     dccpha_ack->dha_ack_high = ack_high;
686     dccpha_ack->dha_ack_low = ack_low;
687     dccpha_ack->dha_ack_reserved = 0;

689     dccpha_srv = (dccpha_srv_t *) &mpls->b_rptr[ip_hdr_len + sizeof (dccpha_t)
690     dccpha_srv->dha_srv_code = 0;

692     ixa->ixa_pktlen = total_hdr_len;

694     if (ixa->ixa_flags & IXAF_IS_IPV4) {
695         ((ipha_t *) rptra)->ipha_length = htons(total_hdr_len);
696     } else {
697         ip6_t *ip6 = (ip6_t *) rptra;

699         ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
700     }

702     cmn_err(CE_NOTE, "IPHC LEN: %d", connp->conn_ht_iphc_len);
703     cmn_err(CE_NOTE, "TOTAL LEN: %d", total_hdr_len);

705     kmem_free(options, opt_len);

707     return (mpls);
708 }

710 /*
711  * Generate a request packet. Must use 48-bit sequence
712  * numbers.
713  */
714 mblk_t *
715 dccp_generate_request(conn_t *connp)
716 {
717     dccp_t      *dccp = connp->conn_dccp;
718     dccpha_t    *dccpha;
719     dccpha_ext_t *dccpha_ext;
720     dccpha_srv_t *dccpha_srv;
721     ip_xmit_attr_t *ixa = connp->conn_ixa;

```

```

722     mblk_t      *mp;
723     uchar_t     *rptra;
724     uint_t      total_hdr_len;
725     uint_t      len;

727     cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_request");

729     total_hdr_len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
730     sizeof (dccpha_srv_t);
731     mp = allocb(total_hdr_len, BPRI_MED);
732     if (mp == NULL) {
733         cmn_err(CE_NOTE, "allocb failed");
734         return (NULL);
735     }

737     rptra = mp->b_rptr;
738     mp->b_wptr = &mp->b_rptr[total_hdr_len];

740     /* Copy in the template header */
741     bcopy(connp->conn_ht_iphc, rptra, connp->conn_ht_iphc_len);

743     len = ixa->ixa_ip_hdr_length;
744     dccpha = (dccpha_t *) &rptra[len];
745     dccpha->dha_type = DCCP_PKT_REQUEST;
746     dccpha->dha_offset = (sizeof (dccpha_t) + sizeof (dccpha_ext_t) +
747     sizeof (dccpha_srv_t)) / 4;
748     dccpha->dha_x = 1;
749     dccpha->dha_ccval = 0;
750     dccpha->dha_cscov = 0;
751     dccpha->dha_reserved = 0;
752     dccpha->dha_res_seq = 0;
753     dccpha->dha_seq = 0;
754     dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) +
755     sizeof (dccpha_srv_t));

757     /* Extended sequence number */
758     len += sizeof (dccpha_t);
759     dccpha_ext = (dccpha_ext_t *) &rptra[len];

761     /* Service number */
762     len += sizeof (dccpha_ext_t);
763     dccpha_srv = (dccpha_srv_t *) &rptra[len];
764     dccpha_srv->dha_srv_code = 0;

766     ixa->ixa_pktlen = total_hdr_len;

768     if (ixa->ixa_flags & IXAF_IS_IPV4) {
769         ((ipha_t *) rptra)->ipha_length = htons(total_hdr_len);
770     } else {
771         ip6_t *ip6 = (ip6_t *) rptra;

773         ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
774     }

776     return (mp);
777 }

779 /*
780  * Close packet.
781  */
782 mblk_t *
783 dccp_generate_reset(conn_t *connp)
784 {
785     dccp_t      *dccp = connp->conn_dccp;
786     dccpha_t    *dccpha;
787     dccpha_ext_t *dccpha_ext;

```

```

788     dccpha_ack_t      *dccpha_ack;
789     dccpha_reset_t     *dccpha_reset;
790     ip_xmit_attr_t     *ixa = connp->conn_ixa;
791     mblk_t             *mp;
792     uint64_t           gss;
793     uchar_t            *rptr;
794     uint_t              total_hdr_len;
795     uint_t              len = ixa->ixa_ip_hdr_length;

797     cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_reset");

799     /* XXX */
800     dccp->dccp_gss++;

802     /*
803      * Allocate a new DCCP reset message
804      */
805     total_hdr_len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
806         sizeof (dccpha_ack_t) + sizeof (dccpha_reset_t);
807     mp = allocb(total_hdr_len, BPRI_MED);
808     if (mp == NULL) {
809         cmn_err(CE_NOTE, "allocb failed");
810         return(NULL);
811     }

813     rptr = mp->b_rptr;
814     mp->b_wptr = &mp->b_rptr[total_hdr_len];

816     bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);

818     len = ixa->ixa_ip_hdr_length;
819     dccpha = (dccpha_t *) &mp->b_rptr[len];
820     dccpha->dha_type = DCCP_PKT_RESET;
821     dccpha->dha_offset = 7;
822     dccpha->dha_x = 1;
823     dccpha->dha_ccval = 0;
824     dccpha->dha_cscov = 0;
825     dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) +
826         sizeof (dccpha_ack_t) + sizeof (dccpha_reset_t));
827     dccpha->dha_seq = htons(dccp->dccp_gss >> 32);
828     dccpha->dha_res_seq = 0;

830     len += sizeof (dccpha_t);
831     dccpha_ext = (dccpha_ext_t *) &mp->b_rptr[len];
832     dccpha_ext->dha_ext_seq = htonl(dccp->dccp_gss & 0xffffffff);

834     /* Set ack */
835     len += sizeof (dccpha_ext_t);
836     dccpha_ack = (dccpha_ack_t *) &mp->b_rptr[len];
837     dccpha_ack->dha_ack_high = htons(dccp->dccp_gsr >> 32);
838     dccpha_ack->dha_ack_low = htonl(dccp->dccp_gsr & 0xffffffff);
839     dccpha_ack->dha_ack_reserved = 0;

841     len += sizeof (dccpha_ack_t);
842     dccpha_reset = (dccpha_reset_t *) &mp->b_rptr[len];
843     dccpha_reset->dha_reset_code = dccp->dccp_reset_code;
844     dccpha_reset->dha_reset_data[0] = dccp->dccp_reset_data[0];
845     dccpha_reset->dha_reset_data[1] = dccp->dccp_reset_data[1];
846     dccpha_reset->dha_reset_data[2] = dccp->dccp_reset_data[2];

848     ixa->ixa_pktlen = total_hdr_len;

850     if (ixa->ixa_flags & IXAF_IS_IPV4) {
851         cmn_err(CE_NOTE, "setting ip len for ipv4: %d", total_hdr_len);
852         ((ipha_t *) rptr)->ipha_length = htons(total_hdr_len);
853     } else {

```

```

854         ip6_t *ip6 = (ip6_t *) rptr;

856         ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
857     }

859     return (mp);
860 }

862 /*
863  * Acknowledgement packet.
864  */
865     mblk_t *
866     dccp_generate_ack(conn_t *connp)
867     {
868         dccp_t          *dccp = connp->conn_dccp;
869         dccpha_t         *dccpha;
870         dccpha_ext_t     *dccpha_ext;
871         dccpha_ack_t     *dccpha_ack;
872         ip_xmit_attr_t   *ixa = connp->conn_ixa;
873         mblk_t           *mp;
874         uint64_t          gss;
875         uchar_t           *rprior;
876         uint_t             total_hdr_len;
877         uint_t             len = ixa->ixa_ip_hdr_length;

879         cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_ack");

881         /* XXX */
882         dccp->dccp_gss++;

884         /*
885          * Allocate a new DCCP reset message
886          */
887         total_hdr_len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
888             sizeof (dccpha_ack_t);
889         mp = allocb(total_hdr_len, BPRI_MED);
890         if (mp == NULL) {
891             cmn_err(CE_NOTE, "allocb failed");
892             return(NULL);
893         }

895         rptr = mp->b_rptr;
896         mp->b_wptr = &mp->b_rptr[total_hdr_len];

898         bcopy(connp->conn_ht_iphc, rptr, connp->conn_ht_iphc_len);

900         len = ixa->ixa_ip_hdr_length;
901         dccpha = (dccpha_t *) &mp->b_rptr[len];
902         dccpha->dha_type = DCCP_PKT_ACK;
903         dccpha->dha_offset = 6;
904         dccpha->dha_x = 1;
905         dccpha->dha_ccval = 0;
906         dccpha->dha_cscov = 0;
907         dccpha->dha_sum = htons(sizeof (dccpha_t) + sizeof (dccpha_ext_t) +
908             sizeof (dccpha_ack_t));
909         dccpha->dha_seq = htons(dccp->dccp_gss >> 32);
910         dccpha->dha_res_seq = 0;

912         len += sizeof (dccpha_t);
913         dccpha_ext = (dccpha_ext_t *) &mp->b_rptr[len];
914         dccpha_ext->dha_ext_seq = htonl(dccp->dccp_gss & 0xffffffff);

916         len += sizeof (dccpha_ext_t);
917         dccpha_ack = (dccpha_ack_t *) &mp->b_rptr[len];
918         dccpha_ack->dha_ack_high = htons(dccp->dccp_gsr >> 32);
919         dccpha_ack->dha_ack_low = htonl(dccp->dccp_gsr & 0xffffffff);

```



```

920     dccpha_ack->dha_ack_reserved = 0;

922     ixa->ixa_pktlen = total_hdr_len;

924     if (ixa->ixa_flags & IXAF_IS_IPV4) {
925         ((ipha_t *)rptra->ipha_length = htons(total_hdr_len);
926     } else {
927         ip6_t *ip6 = (ip6_t *)rptra;

929         ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
930     }

932     return (mp);
933 }

935 /*
936  * Common to all packet generating functions.
937  */
938 static mblk_t *
939 dccp_generate_common(conn_t *connp, uint8_t packet_type,
940     uint_t total_hdr_len)
941 {
942     dccp_t      *dccp = connp->conn_dccp;
943     dccpha_t    *dccpha;
944     dccpha_ext_t *dccpha_ext;
945     dccpha_ack_t *dccpha_ack;
946     ip_xmit_attr_t *ixa = connp->conn_ixa;
947     mblk_t      *mp;
948     uchar_t     *rptra;
949     uint_t      len;
950     boolean_t    extended;

952     /*
953      * Allocate a new DCCP reset message
954      */
955     mp = allocb(total_hdr_len, BPRI_MED);
956     if (mp == NULL) {
957         return(NULL);
958     }

960     dccp->dccp_gss++;

962     rptra = mp->b_rptra;
963     mp->b_wptra = &mp->b_rptra[total_hdr_len];

965     bcopy(connp->conn_ht_iphc, rptra, connp->conn_ht_iphc_len);

967     /* Basic header */
968     len = ixa->ixa_ip_hdr_length;
969     dccpha = (dccpha_t *)&mp->b_rptra[len];
970     dccpha->dha_type = packet_type;
971     dccpha->dha_offset = 6;
972     dccpha->dha_x = 1;
973     dccpha->dha_ccval = 0;
974     dccpha->dha_cscov = 0;
975     dccpha->dha_sum = htons(total_hdr_len - len);
976     dccpha->dha_seq = htons(dccp->dccp_gss >> 32);
977     dccpha->dha_res_seq = 0;

979     /* Extended sequence number */
980     len += sizeof (dccpha_t);
981     dccpha_ext = (dccpha_ext_t *)&mp->b_rptra[len];
982     dccpha_ext->dha_ext_seq = htonl(dccp->dccp_gss & 0xffffffff);

984     /* Acknowledgement number */
985     len += sizeof (dccpha_ext_t);

```

```

986     dccpha_ack = (dccpha_ack_t *)&mp->b_rptra[len];
987     dccpha_ack->dha_ack_high = htons(dccp->dccp_gsr >> 32);
988     dccpha_ack->dha_ack_low = htonl(dccp->dccp_gsr & 0xffffffff);
989     dccpha_ack->dha_ack_reserved = 0;

991     ixa->ixa_pktlen = total_hdr_len;

993     if (ixa->ixa_flags & IXAF_IS_IPV4) {
994         ((ipha_t *)rptra->ipha_length = htons(total_hdr_len);
995     } else {
996         ip6_t *ip6 = (ip6_t *)rptra;

998         ip6->ip6_plen = htons(total_hdr_len - IPV6_HDR_LEN);
999     }

1001     return (mp);
1002 }

1004 /*
1005  * Acknowledgement packet.
1006  */
1007 mblk_t *
1008 dccp_generate_sync(conn_t *connp)
1009 {
1010     dccp_t      *dccp = connp->conn_dccp;
1011     dccpha_t    *dccpha;
1012     dccpha_ext_t *dccpha_ext;
1013     dccpha_ack_t *dccpha_ack;
1014     ip_xmit_attr_t *ixa = connp->conn_ixa;
1015     mblk_t      *mp;
1016     uchar_t     *rptra;
1017     uint_t      total_hdr_len;
1018     uint_t      len = ixa->ixa_ip_hdr_length;

1020     cmn_err(CE_NOTE, "dccp_output.c: dccp_generate_ack");

1022     total_hdr_len = connp->conn_ht_iphc_len + sizeof (dccpha_ext_t) +
1023         sizeof (dccpha_ack_t);
1024     mp = dccp_generate_common(connp, DCCP_PACKET_SYNC, total_hdr_len);
1025     if (mp == NULL) {
1026         cmn_err(CE_NOTE, "allocb failed");
1027         return(NULL);
1028     }

1030     rptra = mp->b_rptra;
1031     mp->b_wptra = &mp->b_rptra[total_hdr_len];

1033     return (mp);
1034 }
1035 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp/dccp_socket.c

1

```
*****
20359 Sat Aug 18 10:37:10 2012
new/usr/src/uts/common/inet/dccp/dccp_socket.c
dccp: starting module template
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

27 /*
28  * Copyright 2012 David Hoeppner. All rights reserved.
29 */

31 /*
32  * This file contains all DCCP kernel socket related functions.
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/timod.h>
44 #include <sys/squeue_impl.h>
45 #include <sys/squeue.h>
46 #include <sys/socketvar.h>

48 #include <inet/common.h>
49 #include <inet/dccp_impl.h>
50 #include <inet/dccp_stack.h>
51 #include <inet/proto_set.h>
52 #include <inet/ip.h>

54 #include <sys/cmn_err.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 *);
```

new/usr/src/uts/common/inet/dccp/dccp_socket.c

2

```
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_send(sock_lower_handle_t, mblk_t *, struct nmsghdr *,
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, /* sd_activate */
83     dccp_accept, /* sd_accept */
84     dccp_bind, /* sd_bind */
85     dccp_listen, /* sd_listen */
86     dccp_connect, /* sd_connect */
87     dccp_getpeername, /* sd_getpeername */
88     dccp_getsockname, /* sd_getsockname */
89     dccp_getsockopt, /* sd_getsockopt */
90     dccp_setsockopt, /* sd_setsockopt */
91     dccp_send, /* sd_send */
92     NULL, /* sd_send_uio */
93     NULL, /* sd_recv_uio */
94     NULL, /* sd_poll */
95     dccp_shutdown, /* sd_shutdown */
96     dccp_clr_flowctrl, /* sd_setflowctrl */
97     dccp_ioctl, /* sd_ioctl */
98     dccp_close, /* sd_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 dccp_rinfo;

110     cmn_err(CE_NOTE, "dccp_socket.c: dccp_activate");

112     ASSERT(connp->conn_upper_handle == NULL);

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

117     sopp.sopp_flags = SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT |
118         SOCKOPT_MAXPSZ | SOCKOPT_MAXBLK | SOCKOPT_RCVTIMER |
119         SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ;

121     sopp.sopp_rxhiwat = SOCKET_RECVHIWATER;
122     sopp.sopp_rxlowat = SOCKET_RECVLOWATER;
123     sopp.sopp_maxpsz = INFPSZ;
124     sopp.sopp_maxblk = INFPSZ;
125     sopp.sopp_rcvtimer = SOCKET_TIMER_INTERVAL;
126     sopp.sopp_rcvthresh = SOCKET_RECVHIWATER >> 3;
127     sopp.sopp_maxaddrlen = sizeof(sin6_t);
```

```

128     sopp.sopp_minpsz = (dccp_rinfo.mi_minpsz == 1) ? 0 :
129         dccp_rinfo.mi_minpsz;

131     connp->conn_upcalls = sock_upcalls;
132     connp->conn_upper_handle = sock_handle;

134     /* XXX conn_rcvbuf */
135     (*connp->conn_upcalls->su_set_proto_props)(connp->conn_upper_handle,
136         &sopp);
137 }

139 /*ARGSUSED*/
140 static int
141 dccp_accept(sock_lower_handle_t lproto_handle,
142     sock_lower_handle_t eproto_handle, sock_upper_handle_t sock_handle,
143     cred_t *cr)
144 {
145     conn_t *lconnp, *econnp;
146     dccp_t *listener, *eager;

148     cmn_err(CE_NOTE, "dccp_socket.c: dccp_accept");

150     econnp = (conn_t *)eprotocol_handle;
151     eager = econnp->conn_dccp;
152     ASSERT(IPCL_IS_NONSTR(econnp));
153     ASSERT(eager->dccp_listener != NULL);
154     listener = eager->dccp_listener;
155     lconnp = (conn_t *)listener->dccp_connp;
156     ASSERT(listener->dccp_state == DCCPS_LISTEN);
157     ASSERT(lconnp->conn_upper_handle != NULL);

159     ASSERT(econnp->conn_upper_handle == NULL ||
160         econnp->conn_upper_handle == sock_handle);
161     ASSERT(econnp->conn_upcalls == NULL ||
162         econnp->conn_upcalls == lconnp->conn_upcalls);
163     econnp->conn_upper_handle = sock_handle;
164     econnp->conn_upcalls = lconnp->conn_upcalls;

166     ASSERT(econnp->conn_netstack ==
167         listener->dccp_connp->conn_netstack);
168     ASSERT(eager->dccp_dccps == listener->dccp_dccps);

170     ASSERT(econnp->conn_ref >= 2);
171     eager->dccp_listener = NULL; /* XXX */
172     CONN_DEC_REF(listener->dccp_connp);

174     return ((eager->dccp_state < DCCPS_OPEN) ? ECONNABORTED : 0);
175 }

177 static int
178 dccp_bind(sock_lower_handle_t proto_handle, struct sockaddr *sa,
179     socklen_t len, cred_t *cr)
180 {
181     conn_t *connp = (conn_t *)proto_handle;
182     int error;

184     cmn_err(CE_NOTE, "dccp_socket.c: dccp_bind");

186     ASSERT(connp->conn_upper_handle != NULL);

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

191     error = queue_synch_enter(connp, NULL);
192     if (error != 0) {
193         /* Failed to enter */

```

```

194         return (ENOSR);
195     }

197     /* Binding to NULL address means unbind */
198     if (sa == NULL) {
199         if (connp->conn_dccp->dccp_state < DCCPS_LISTEN) {
200             error = dccp_do_unbind(connp);
201         } else {
202             error = EINVAL;
203         }
204     } else {
205         error = dccp_do_bind(connp, sa, len, cr, B_TRUE);
206     }

208     queue_synch_exit(connp);

210     if (error < 0) {
211         if (error == -TOUTSTATE) {
212             error = EINVAL;
213         } else {
214             error = proto_tltosyserr(-error);
215         }
216     }

218     return (error);
219 }

221 /* ARGSUSED */
222 static int
223 dccp_listen(sock_lower_handle_t proto_handle, int backlog, cred_t *cr)
224 {
225     conn_t *connp = (conn_t *)proto_handle;
226     dccp_t *dccp = connp->conn_dccp;
227     int error;

229     cmn_err(CE_NOTE, "dccp_socket.c: dccp_listen");

231     ASSERT(connp->conn_upper_handle != NULL);

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

236     error = queue_synch_enter(connp, NULL);
237     if (error != 0) {
238         /* Failed to enter */
239         return (ENOBUFS);
240     }

242     error = dccp_do_listen(connp, NULL, 0, backlog, cr, B_FALSE);
243     if (error == 0) {
244         /* XXX dccps->dccps_conn_req_max_q0 */
245         (*connp->conn_upcalls->su_opctl)(connp->conn_upper_handle,
246             SOCK_OPCTL_ENAB_ACCEPT,
247             (uintptr_t)(dccp->dccp_conn_req_max));
248     } else if (error < 0) {
249         if (error == -TOUTSTATE) {
250             error = EINVAL;
251         } else {
252             error = proto_tltosyserr(-error);
253         }
254     }

256     queue_synch_exit(connp);

258     return (error);
259 }

```

```

261 /*
262  * Socket connect.
263  */
264 static int
265 dccp_connect(sock_lower_handle_t proto_handle, const struct sockaddr *sa,
266             socklen_t len, sock_connid_t *id, cred_t *cr)
267 {
268     conn_t *connp = (conn_t *)proto_handle;
269     int error;
270
271     cmn_err(CE_NOTE, "dccp_socket.c: dccp_connect");
272
273     ASSERT(connp->conn_upper_handle != NULL);
274
275     /* All Solaris components should pass a cred for this operation */
276     ASSERT(cr != NULL);
277
278     error = proto_verify_ip_addr(connp->conn_family, sa, len);
279     if (error != 0) {
280         return (error);
281     }
282
283     error = squeue_synch_enter(connp, NULL);
284     if (error != 0) {
285         /* Failed to enter */
286         return (ENOSR);
287     }
288
289     /*
290      * DCCP supports quick connect, so no need to do an implicit bind.
291      */
292     error = dccp_do_connect(connp, sa, len, cr, curproc->p_pid);
293     if (error == 0) {
294         *id = connp->conn_dccp->dccp_connid;
295     } else if (error < 0) {
296         if (error == -TOUTSTATE) {
297             switch (connp->conn_dccp->dccp_state) {
298                 case DCCPS_REQUEST:
299                     error = EALREADY;
300                     break;
301                 case DCCPS_PARTOPEN:
302                     error = EISCONN;
303                     break;
304                 case DCCPS_LISTEN:
305                     error = EOPNOTSUPP;
306                     break;
307                 default:
308                     error = EINVAL;
309                     break;
310             }
311         } else {
312             error = proto_tltosyserr(-error);
313         }
314     }
315
316     /* XXX loopback */
317
318     squeue_synch_exit(connp);
319
320     return ((error == 0) ? EINPROGRESS : error);
321 }
322
323 /* ARGSUSED3 */
324 static int
325 dccp_getpeername(sock_lower_handle_t proto_handle, struct sockaddr *addr,

```

```

326     socklen_t *addrlenp, cred_t *cr)
327 {
328     conn_t *connp = (conn_t *)proto_handle;
329     dccp_t *dccp = connp->conn_dccp;
330
331     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getpeername");
332
333     /* All Solaris components should pass a cred for this operation */
334     ASSERT(cr != NULL);
335
336     ASSERT(dccp != NULL);
337     if (dccp->dccp_state < DCCPS_OPEN) {
338         return (ENOTCONN);
339     }
340
341     return (conn_getpeername(connp, addr, addrlenp));
342 }
343
344 /* ARGSUSED3 */
345 static int
346 dccp_getsockname(sock_lower_handle_t proto_handle, struct sockaddr *addr,
347                 socklen_t *addrlenp, cred_t *cr)
348 {
349     conn_t *connp = (conn_t *)proto_handle;
350     int error;
351
352     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockname");
353
354     /* All Solaris components should pass a cred for this operation */
355     ASSERT(cr != NULL);
356
357     /* XXX UDP has locks here, TCP not */
358     mutex_enter(&connp->conn_lock);
359     error = conn_getsockname(connp, addr, addrlenp);
360     mutex_exit(&connp->conn_lock);
361
362     return (error);
363 }
364
365 static int
366 dccp_getsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
367                void *optvalp, socklen_t *optlen, cred_t *cr)
368 {
369     conn_t *connp = (conn_t *)proto_handle;
370     void *optvalp_buf;
371     t_uscalar_t max_optbuf_len;
372     int len;
373     int error;
374
375     cmn_err(CE_NOTE, "dccp_socket.c: dccp_getsockopt");
376
377     ASSERT(connp->conn_upper_handle != NULL);
378
379     /* All Solaris components should pass a cred for this operation */
380     ASSERT(cr != NULL);
381
382     error = proto_opt_check(level, option_name, *optlen, &max_optbuf_len,
383                             dccp_opt_obj.odt_opt_des_arr,
384                             dccp_opt_obj.odt_opt_arr_cnt,
385                             B_FALSE, B_TRUE, cr);
386     if (error != 0) {
387         if (error < 0) {
388             error = proto_tltosyserr(-error);
389         }
390         return (error);
391     }

```

```

393     optvalp_buf = kmem_alloc(max_optbuf_len, KM_SLEEP);
394     if (optvalp_buf == NULL) {
395         return (ENOMEM);
396     }
398     error = queue_synch_enter(connp, NULL);
399     if (error == ENOMEM) {
400         kmem_free(optvalp_buf, max_optbuf_len);
401         return (ENOMEM);
402     }
404     len = dccp_opt_get(connp, level, option_name, optvalp_buf);
405     queue_synch_exit(connp);
407     if (len == -1) {
408         kmem_free(optvalp_buf, max_optbuf_len);
409         return (EINVAL);
410     }
412     /*
413      * Update optlen and copy option value.
414      */
415     t_uscalar_t size = MIN(len, *optlen);
417     bcopy(optvalp_buf, optvalp, size);
418     bcopy(&size, optlen, sizeof (size));
420     kmem_free(optvalp_buf, max_optbuf_len);
422     return (0);
423 }
425 static int
426 dccp_setsockopt(sock_lower_handle_t proto_handle, int level, int option_name,
427     const void *optvalp, socklen_t optlen, cred_t *cr)
428 {
429     conn_t *connp = (conn_t *)proto_handle;
430     int error;
432     cmn_err(CE_NOTE, "dccp_socket.c: dccp_setsockopt");
434     ASSERT(connp->conn_upper_handle != NULL);
436     /* All Solaris components should pass a cred for this operation */
437     ASSERT(cr != NULL);
439     error = queue_synch_enter(connp, NULL);
440     if (error == ENOMEM) {
441         return (ENOMEM);
442     }
444     error = proto_opt_check(level, option_name, optlen, NULL,
445         dccp_opt_obj.odb_opt_des_arr,
446         dccp_opt_obj.odb_opt_arr_cnt,
447         B_TRUE, B_FALSE, cr);
449     if (error != 0) {
450         if (error < 0) {
451             error = proto_tltosyserr(-error);
452         }
453         queue_synch_exit(connp);
454         return (error);
455     }
457     error = dccp_opt_set(connp, SETFN_OPTCOM_NEGOTIATE, level, option_name,

```

```

458         optlen, (uchar_t *)optvalp, (uint_t *)&optlen, (uchar_t *)optvalp,
459         NULL, cr);
460     queue_synch_exit(connp);
462     ASSERT(error >= 0);
464     return (error);
465 }
467 /* ARGSUSED */
468 static int
469 dccp_send(sock_lower_handle_t proto_handle, mblk_t *mp, struct nmsg_hdr *msg,
470     cred_t *cr)
471 {
472     conn_t *connp = (conn_t *)proto_handle;
473     dccp_t *dccp;
474     uint32_t msize;
475     int32_t dccpstate;
477     cmn_err(CE_NOTE, "dccp_socket.c: dccp_send");
479     /* All Solaris components should pass a cred for this operation */
480     ASSERT(cr != NULL);
482     ASSERT(connp->conn_ref >= 2);
483     ASSERT(connp->conn_upper_handle != NULL);
485     if (msg->msg_controllen != 0) {
486         freemsg(mp);
487         return (EOPNOTSUPP);
488     }
490     switch (DB_TYPE(mp)) {
491     case M_DATA:
492         dccp = connp->conn_dccp;
493         ASSERT(dccp != NULL);
495         dccpstate = dccp->dccp_state;
496         if (dccpstate < DCCPS_OPEN) {
497             freemsg(mp);
499             /*
500              * We return ENOTCONN if the endpoint is trying to
501              * connect or has never been connected, and EPIPE if it
502              * has been disconnected. The connection id helps us
503              * distinguish between the last two cases.
504              */
505             return ((dccpstate == DCCPS_REQUEST) ? ENOTCONN :
506                 ((dccp->dccp_connid > 0) ? EPIPE : ENOTCONN));
507         } else if (dccpstate > DCCPS_CLOSING) {
508             freemsg(mp);
509             return (EPIPE);
510         }
512         /* XXX */
514         msize = msgdsize(mp);
516         CONN_INC_REF(connp);
518         if (msg->msg_flags & MSG_OOB) {
519             QUEUE_ENTER_ONE(connp->conn_sq, mp, dccp_output_urgent,
520                 connp, NULL, dccp_queue_flag, SQTAG_DCCP_OUTPUT);
521         } else {
522             QUEUE_ENTER_ONE(connp->conn_sq, mp, dccp_output,
523                 connp, NULL, dccp_queue_flag, SQTAG_DCCP_OUTPUT);

```

```

524     }
525
526     return (0);
527
528     default:
529         ASSERT(0);
530     }
531
532     freemsg(mp);
533
534     return (0);
535 }
536
537 /* ARGSUSED */
538 static int
539 dccp_shutdown(sock_lower_handle_t proto_handle, int how, cred_t *cr)
540 {
541     conn_t *connp = (conn_t *)proto_handle;
542     dccp_t *dccp = connp->conn_dccp;
543
544     cmn_err(CE_NOTE, "dccp_socket.c: dccp_shutdown");
545
546     /* All Solaris components should pass a cred for this operation. */
547     ASSERT(cr != NULL);
548
549     ASSERT(connp->conn_upper_handle != NULL);
550
551     /*
552      * X/Open requires that we check the connected state.
553      */
554     if (dccp->dccp_state < DCCPS_REQUEST)
555         return (ENOTCONN);
556
557     /* Shutdown the send side */
558     if (how != SHUT_RD) {
559         mblk_t *bp;
560
561         bp = allocb_wait(0, BPRI_HI, STR_NOSIG, NULL);
562         CONN_INC_REF(connp);
563         SQUEUE_ENTER_ONE(connp->conn_sq, bp, dccp_shutdown_output,
564             connp, NULL, SQ_NODRAIN, SQTAG_DCCP_SHUTDOWN_OUTPUT);
565
566         (*connp->conn_upcalls->su_opctl)(connp->conn_upper_handle,
567             SOCK_OPCTL_SHUT_SEND, 0);
568     }
569
570     /* Shutdown the recv side */
571     if (how != SHUT_WR) {
572         (*connp->conn_upcalls->su_opctl)(connp->conn_upper_handle,
573             SOCK_OPCTL_SHUT_RECV, 0);
574     }
575
576     return (0);
577 }
578
579 static void
580 dccp_clr_flowctrl(sock_lower_handle_t proto_handle)
581 {
582     conn_t *connp = (conn_t *)proto_handle;
583     dccp_t *dccp = connp->conn_dccp;
584     mblk_t *mp;
585     int error;
586
587     ASSERT(connp->conn_upper_handle != NULL);
588
589     cmn_err(CE_NOTE, "dccp_socket.c: dccp_clr_flowctrl");

```

```

591     error = squeue_synch_enter(connp, mp);
592     ASSERT(error == 0);
593
594     squeue_synch_exit(connp);
595 }
596
597 /* ARGSUSED */
598 static int
599 dccp_ioctl(sock_lower_handle_t proto_handle, int cmd, intptr_t arg,
600     int mode, int32_t *rvalp, cred_t *cr)
601 {
602     conn_t *connp = (conn_t *)proto_handle;
603     int error;
604
605     cmn_err(CE_NOTE, "dccp_socket.c: dccp_ioctl");
606
607     ASSERT(connp->conn_upper_handle != NULL);
608
609     /* All Solaris components should pass a cred for this operation. */
610     ASSERT(cr != NULL);
611
612     /*
613      * If we don't have a helper stream then create one.
614      * ip_create_helper_stream takes care of locking the conn_t,
615      * so this check for NULL is just a performance optimization.
616      */
617     if (connp->conn_helper_info == NULL) {
618         dccp_stack_t *dccps = connp->conn_dccp->dccp_dccps;
619
620         /*
621          * Create a helper stream for non-STREAMS socket.
622          */
623         error = ip_create_helper_stream(connp, dccps->dccps_ldi_ident);
624         if (error != 0) {
625             ip0dbg(("dccp_ioctl: create of IP helper stream "
626                 "failed %d\n", error));
627             return (error);
628         }
629     }
630
631     switch (cmd) {
632     case ND_SET:
633     case ND_GET:
634     case _SIOCSOCKFALLBACK:
635     /* XXX case DCCP_IOC_ABORT_CONN: */
636     case TI_GETPEERNAME:
637     case TI_GETMYNAME:
638         ip0dbg(("dccp_ioctl: cmd 0x%x on non streams socket", cmd));
639         error = EINVAL;
640         break;
641     default:
642         /*
643          * If the conn is not closing, pass on to IP using
644          * helper stream. Bump the iocltref to prevent dccp_close
645          * from closing the rq/wq out from underneath the ioctl
646          * if it ends up queued or aborted/interrupted.
647          */
648         mutex_enter(&connp->conn_lock);
649         if (connp->conn_state_flags & (CONN_CLOSING)) {
650             mutex_exit(&connp->conn_lock);
651             error = EINVAL;
652             break;
653         }
654         CONN_INC_IOCTLREF_LOCKED(connp);
655         error = ldi_ioctl(connp->conn_helper_info->iphs_handle,

```

```

656         cmd, arg, mode, cr, rvalp);
657         CONN_DEC_IOCTLREF(connp);
658         break;
659     }
661     return (error);
662 }

664 /* ARGSUSED */
665 static int
666 dccp_close(sock_lower_handle_t proto_handle, int flags, cred_t *cr)
667 {
668     conn_t *connp = (conn_t *)proto_handle;

670     cmn_err(CE_NOTE, "dccp_socket.c: dccp_close\n");

672     ASSERT(connp->conn_upper_handle != NULL);

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

677     dccp_close_common(connp, flags);

679     ip_free_helper_stream(connp);

681     CONN_DEC_REF(connp);

683     /*
684      * EINPROGRESS tells sockfs to wait for a 'closed' upcall before
685      * freeing the socket.
686      */
687     return (EINPROGRESS);
688 }

691 /*
692  * Socket create function.
693  */
694 sock_lower_handle_t
695 dccp_create(int family, int type, int proto, sock_downcalls_t **sockdowncalls,
696            uint_t *smodep, int *errorp, int flags, cred_t *credp)
697 {
698     conn_t *connp;
699     boolean_t isv6;

701     /* XXX (type != SOCK_STREAM */
702     if ((family != AF_INET && family != AF_INET6) ||
703         (proto != 0 && proto != IPPROTO_DCCP)) {
704         *errorp = EPROTONOSUPPORT;
705         return (NULL);
706     }

708     cmn_err(CE_NOTE, "dccp_socket: dccp_create\n");

710     isv6 = family == AF_INET6 ? B_TRUE: B_FALSE;
711     connp = dccp_create_common(credp, isv6, B_TRUE, errorp);
712     if (connp == NULL) {
713         return (NULL);
714     }

716     /*
717      * Increment ref for DCCP connection.
718      */
719     mutex_enter(&connp->conn_lock);
720     CONN_INC_REF_LOCKED(connp);
721     ASSERT(connp->conn_ref == 2);

```

```

722     connp->conn_state_flags &= ~CONN_INCIPIENT;
723     connp->conn_flags |= IPCL_NONSTR;
724     mutex_exit(&connp->conn_lock);

726     ASSERT(errorp != NULL);
727     *errorp = 0;
728     *sockdowncalls = &sock_dccp_downcalls;
729     *smodep = SM_CONNREQUIRED | SM_EXDATA | SM_ACCEPTSUPP |
730             SM_SENDFILESUPP;

732     return ((sock_lower_handle_t)connp);
733 }

735 int
736 dccp_fallback(sock_lower_handle_t proto_handle, queue_t *q,
737              boolean_t issocket, so_proto_quiesced_cb_t quiesced_cb,
738              sock_quiesce_arg_t *arg)
739 {
740     cmn_err(CE_NOTE, "dccp_socket: dccp_fallback\n");

742     return (0);
743 }

745 /*
746  * Notifies a non-STREAMS based listener about a new connection. This
747  * function is executed on the *eager*'s squeue once the 3 way handshake
748  * has completed. Note that the behavior differs from STREAMS, where the
749  * T_CONN_IND is sent up by tcp_send_conn_ind() while on the *listener*'s
750  * squeue.
751  *
752  * Returns B_TRUE if the notification succeeded and an upper handle was
753  * obtained. 'tcp' should be closed on failure.
754  */
755 boolean_t
756 dccp_newconn_notify(dccp_t *dccp, ip_rcv_attr_t *ira)
757 {
758     dccp_t *listener = dccp->dccp_listener;
759     dccp_t *tail;
760     conn_t *lconnp = listener->dccp_connnp;
761     conn_t *econnp = dccp->dccp_connnp;
762     ipaddr_t *addr_cache;
763     sock_upper_handle_t upper;
764     struct sock_proto_props sopp;

766     cmn_err(CE_NOTE, "dccp_socket.c: dccp_newconn_notify");

768     /*
769      * mutex_enter(&listener->dccp_eager_lock);
770      * addr_cache = (ipaddr_t *) (listener->dccp_ip_addr_cache);
771      * if (addr_cache != NULL) {
772      *     addr_cache[IP_ADDR_CACHE_HASH(dccp->dccp_connnp->conn_faddr_v4)]
773      *     dccp->dccp_connnp->conn_faddr_v4;
774      * }
775      * mutex_exit(&listener->dccp_eager_lock);
776      */

778     /*
779      * Notify the ULP about the new connection.
780      */
781     if ((upper = (*lconnp->conn_upcalls->su_newconn)
782              (lconnp->conn_upper_handle, (sock_lower_handle_t) econnp,
783               &sock_dccp_downcalls, ira->ira_cred, ira->ira_cpuid,
784               &econnp->conn_upcalls)) == NULL) {
785         return (B_FALSE);
786     }
787     econnp->conn_upper_handle = upper;

```

```
789     dccp->dccp_detached = B_FALSE;
790     dccp->dccp_hard_binding = B_FALSE;
791     dccp->dccp_tconnind_started = B_TRUE;

793     if (econnp->conn_keepalive) {
794         dccp->dccp_ka_last_intrvl = 0;
795         dccp->dccp_ka_tid = DCCP_TIMER(dccp, dccp_keepalive_timer,
796             dccp->dccp_ka_interval);
797     }

799     /* Update the necessary parameters */
800     dccp_get_proto_props(dccp, &sopp);

802     (*econnp->conn_upcalls->su_set_proto_props)
803         (econnp->conn_upper_handle, &sopp);

805     return (B_TRUE);
806 }
807 #endif /* ! codereview */
```


new/usr/src/uts/common/inet/dccp/dccp_stats.c

1

```
*****
13741 Sat Aug 18 10:37:10 2012
new/usr/src/uts/common/inet/dccp/dccp_stats.c
dccp: MIB-II
*****

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 Hoeppner. All rights reserved.
29 */

31 /*
32  * Functions related to MIB-II and kstat.
33 */

35 #include <sys/types.h>
36 #include <sys/tihdr.h>
37 #include <sys/policy.h>
38 #include <sys/tsol/tnet.h>

40 #include <inet/common.h>
41 #include <inet/dccp_impl.h>
42 #include <inet/ip.h>
43 #include <inet/kstatcom.h>
44 #include <inet/snmpcom.h>

46 #include <sys/cmn_err.h>

48 static int      dccp_snmp_state(dccp_t *);
49 static int      dccp_kstat_update(kstat_t *, int);
50 static int      dccp_kstat2_update(kstat_t *, int);
51 static void      dccp_add_mib(mib2_dccp_t *, mib2_dccp_t *);
52 static void      dccp_sum_mib(dccp_stack_t *, mib2_dccp_t *);
53 static void      dccp_clr_stats(dccp_stat_t *);
54 static void      dccp_add_stats(dccp_stat_counter_t *, dccp_stat_t *);

56 /*
57  * Translate DCCP state to MIB2 state.
58 */
59 static int
60 dccp_snmp_state(dccp_t *dccp)
61 {
```

new/usr/src/uts/common/inet/dccp/dccp_stats.c

2

```
62     if (dccp == NULL) {
63         return (0);
64     }

66     switch(dccp->dccp_state) {
67     case DCCPS_CLOSED:
68         return (MIB2_DCCP_closed);
69     default:
70         return (0);
71     }
72 }

74 /*
75  * Get the MIB-II stats.
76 */
77 mblk_t *
78 dccp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
79 {
80     conn_t          *connp = Q_TO_CONN(q);
81     connf_t          *connfp;
82     ip_stack_t       *ipst;
83     dccp_stack_t     *dccps;
84     struct opthdr    *optp;
85     mblk_t           *mp2ctl;
86     mblk_t           *mpdata;
87     mblk_t           *mp_conn_ctl = NULL;
88     mblk_t           *mp_conn_tail;
89     mblk_t           *mp_attr_ctl = NULL;
90     mblk_t           *mp_attr_tail;
91     mblk_t           *mp6_conn_ctl = NULL;
92     mblk_t           *mp6_conn_tail;
93     mblk_t           *mp6_attr_ctl = NULL;
94     mblk_t           *mp6_attr_tail;
95     size_t           dccp_mib_size;
96     size_t           dce_size;
97     size_t           dce6_size;
98     boolean_t        ispriv;
99     zoneid_t         zoneid;
100    int                v4_conn_idx;
101    int                v6_conn_idx;
102    int                i;
103    mib2_dccp_t        dccp_mib;
104    mib2_dccpConnEntry_t dce;
105    mib2_dccp6ConnEntry_t dce6;
106    mib2_transportMLPEEntry_t mlp;

108    /*
109     * Make a copy of the original message.
110     */
111    mp2ctl = copymsg(mpctl);

113    cmn_err(CE_NOTE, "dccp_stats.c: dccp_snmp_get");

115    if (mpctl == NULL ||
116        (mpdata = mpctl->b_cont) == NULL ||
117        (mp_conn_ctl = copymsg(mpctl)) == NULL ||
118        (mp_attr_ctl = copymsg(mpctl)) == NULL ||
119        (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
120        (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
121        freemsg(mp_conn_ctl);
122        freemsg(mp_attr_ctl);
123        freemsg(mp6_conn_ctl);
124        freemsg(mp6_attr_ctl);
125        freemsg(mpctl);
126        freemsg(mp2ctl);
127        return (NULL);
128    }
```

```

128     }
129
130     ipst = connp->conn_netstack->netstack_ip;
131     dccps = connp->conn_netstack->netstack_dccp;
132
133     if (legacy_req) {
134         dccp_mib_size = LEGACY_MIB_SIZE(&dccp_mib, mib2_dccp_t);
135         dce_size = LEGACY_MIB_SIZE(&dce, mib2_dccpConnEntry_t);
136         dce6_size = LEGACY_MIB_SIZE(&dce6, mib2_dccp6ConnEntry_t);
137     } else {
138         dccp_mib_size = sizeof (mib2_dccp_t);
139         dce_size = sizeof (mib2_dccpConnEntry_t);
140         dce6_size = sizeof (mib2_dccp6ConnEntry_t);
141     }
142
143     bzero(&dccp_mib, sizeof (dccp_mib));
144
145     ispriv = secpolicy_ip_config((Q_TO_CONN(q))>conn_cred, B_TRUE) == 0;
146     zoneid = Q_TO_CONN(q)>conn_zoneid;
147
148     v4_conn_idx = v6_conn_idx = 0;
149     mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
150
151     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
152         ipst = dccps->dccps_netstack->netstack_ip;
153
154         connfp = &ipst->ips_ipcl_globalhash_fanout[i];
155         connp = NULL;
156
157         while ((connp = ipcl_get_next_conn(connfp, connp,
158             IPCL_DCCPCONN)) != NULL) {
159             dccp_t      *dccp;
160             boolean_t   needattr;
161
162             if (connp->conn_zoneid != zoneid) {
163                 continue; /* Not in this zone */
164             }
165
166             dccp = connp->conn_dccp;
167             DCCPS_UPDATE_MIB(dccps, dccpHCInSegs, dccp->dccp_ibsegs)
168             dccp->dccp_ibsegs = 0;
169             DCCPS_UPDATE_MIB(dccps, dccpHCOutSegs, dccp->dccp_obsegs)
170             dccp->dccp_obsegs = 0;
171
172             dce.dccpConnState = dccp_snmp_state(dccp);
173
174             needattr = B_FALSE;
175             bzero(&mlp, sizeof (mlp));
176             if (connp->conn_mlp_type != mlptSingle) {
177                 if (connp->conn_mlp_type == mlptShared ||
178                     connp->conn_mlp_type == mlptBoth) {
179                     mlp.tme_flags |= MIB2_TMEF_SHARED;
180                 }
181
182                 if (connp->conn_mlp_type == mlptPrivate ||
183                     connp->conn_mlp_type == mlptBoth) {
184                     mlp.tme_flags |= MIB2_TMEF_PRIVATE;
185                 }
186
187                 needattr = B_TRUE;
188             }
189
190             if (connp->conn_anon_mlp) {
191                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
192                 needattr = B_TRUE;
193             }

```

```

195         switch (connp->conn_mac_mode) {
196             case CONN_MAC_DEFAULT:
197                 break;
198             case CONN_MAC_AWARE:
199                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
200                 needattr = B_TRUE;
201                 break;
202             case CONN_MAC_IMPLICIT:
203                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
204                 needattr = B_TRUE;
205                 break;
206         }
207
208         if (connp->conn_ixa->ixa_tsl != NULL) {
209             ts_label_t    *tsl;
210
211             tsl = connp->conn_ixa->ixa_tsl;
212             mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
213             mlp.tme_doi = label2doi(tsl);
214             mlp.tme_label = *label2bslabel(tsl);
215             needattr = B_TRUE;
216         }
217
218         /* Create a message to report on IPv6 entries */
219         if (connp->conn_ipversion == IPV6_VERSION) {
220             dce6.dccp6ConnLocalAddress =
221                 connp->conn_laddr_v6;
222             dce6.dccp6ConnRemAddress =
223                 connp->conn_faddr_v6;
224             dce6.dccp6ConnLocalPort =
225                 ntohs(connp->conn_lport);
226             dce6.dccp6ConnRemPort =
227                 ntohs(connp->conn_fport);
228
229             if (connp->conn_ixa->ixa_flags &
230                 IXAF_SCOPEID_SET) {
231                 dce6.dccp6ConnIfIndex =
232                     connp->conn_ixa->ixa_scopeid;
233             } else {
234                 dce6.dccp6ConnIfIndex =
235                     connp->conn_bound_if;
236             }
237
238             /* XXX */
239
240             dce6.dccp6ConnEntryInfo.ce_state =
241                 dccp->dccp_state;
242
243             dce6.dccp6ConnCreationProcess =
244                 (connp->conn_cpuid < 0) ?
245                     MIB2_UNKNOWN_PROCESS : connp->conn_cpuid;
246             dce6.dccp6ConnCreationTime =
247                 connp->conn_open_time;
248
249             (void) snmp_append_data2(mp6_conn_ctl->b_cont,
250                 &mp6_conn_tail, (char *)&dce6, dce6_size);
251
252             mlp.tme_connidx = v6_conn_idx++;
253             if (needattr) {
254                 (void) snmp_append_data2(
255                     mp6_attr_ctl->b_cont,
256                     &mp6_attr_tail, (char *)&mlp,
257                     sizeof (mlp));
258             }
259         }

```

```

261         if (connp->conn_ipversion == IPV4_VERSION ||
262             (dccp->dccp_state <= DCCPS_LISTEN &&
263              !connp->conn_ipv6_v6only &&
264              IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
265
266             if (connp->conn_ipversion == IPV6_VERSION) {
267                 dce.dccpConnRemAddress = INADDR_ANY;
268                 dce.dccpConnLocalAddress = INADDR_ANY;
269             } else {
270                 dce.dccpConnRemAddress =
271                     connp->conn_faddr_v4;
272                 dce.dccpConnLocalAddress =
273                     connp->conn_laddr_v4;
274             }
275
276             dce.dccpConnLocalPort =
277                 ntohs(connp->conn_lport);
278             dce.dccpConnRemPort =
279                 ntohs(connp->conn_fport);
280
281             /* XXX */
282
283             dce.dccpConnEntryInfo.ce_state =
284                 dccp->dccp_state;
285
286             dce.dccpConnCreationProcess =
287                 (connp->conn_cpuid < 0) ?
288                     MIB2_UNKNOWN_PROCESS : connp->conn_cpuid;
289             dce.dccpConnCreationTime =
290                 connp->conn_open_time;
291
292             (void) snmp_append_data2(mp_conn_ctl->b_cont,
293                                     &mp_conn_tail, (char *)&dce, dce_size);
294
295             mlp.tme_connidix = v4_conn_idx++;
296             if (needattr) {
297                 (void) snmp_append_data2(
298                     mp_attr_ctl->b_cont,
299                     &mp_attr_tail, (char *)&mlp,
300                     sizeof(mlp));
301             }
302         }
303     }
304 }
305
306 /* Sum up per CPU stats */
307 dccp_sum_mib(dccps, &dccp_mib);
308
309 /* Fixed length structure for IPv4 and IPv6 counters */
310 SET_MIB(dccp_mib.dccpConnTableSize, dce_size);
311 SET_MIB(dccp_mib.dccp6ConnTableSize, dce6_size);
312
313 /* Synchronize 32- and 64-bit counters */
314 SYNC32_MIB(&dccp_mib, dccpInSegs, dccpHCInSegs);
315 SYNC32_MIB(&dccp_mib, dccpOutSegs, dccpHCOutSegs);
316
317 optp = (struct opthdr *)&mpctl->b_rptr[ sizeof (struct T_optmgmt_ack) ];
318 optp->level = MIB2_DCCP;
319 optp->name = 0;
320 (void) snmp_append_data(mpdata, (char *)&dccp_mib, dccp_mib_size);
321 optp->len = msgdsize(mpdata);
322 qreply(q, mpctl);
323
324 optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
325     sizeof (struct T_optmgmt_ack)];

```

```

326     optp->level = MIB2_DCCP;
327     optp->name = MIB2_DCCP_CONN;
328     optp->len = msgdsize(mp_conn_ctl->b_cont);
329     qreply(q, mp_conn_ctl);
330
331     optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
332         sizeof (struct T_optmgmt_ack)];
333     optp->level = MIB2_DCCP;
334     optp->name = EXPER_XPORT_MLP;
335     optp->len = msgdsize(mp_attr_ctl->b_cont);
336     if (optp->len == 0) {
337         freemsg(mp_attr_ctl);
338     } else {
339         qreply(q, mp_attr_ctl);
340     }
341
342     optp = (struct opthdr *)&mp6_conn_ctl->b_rptr[
343         sizeof (struct T_optmgmt_ack)];
344     optp->level = MIB2_DCCP6;
345     optp->name = MIB2_DCCP6_CONN;
346     optp->len = msgdsize(mp6_conn_ctl->b_cont);
347     qreply(q, mp6_conn_ctl);
348
349     optp = (struct opthdr *)&mp6_attr_ctl->b_rptr[
350         sizeof (struct T_optmgmt_ack)];
351     optp->level = MIB2_DCCP6;
352     optp->name = EXPER_XPORT_MLP;
353     optp->len = msgdsize(mp6_attr_ctl->b_cont);
354     if (optp->len == 0) {
355         freemsg(mp6_attr_ctl);
356     } else {
357         qreply(q, mp6_attr_ctl);
358     }
359
360     return (mp2ctl);
361 }
362
363 /*
364  * DCCP kernel statistics.
365  */
366 void *
367 dccp_kstat_init(netstackid_t stackid)
368 {
369     kstat_t *ksp;
370
371     dccp_named_kstat_t template = {
372         { "activeOpens",      KSTAT_DATA_UINT32, 0 },
373         { "passiveOpens",     KSTAT_DATA_UINT32, 0 },
374         { "inSegs",           KSTAT_DATA_UINT64, 0 },
375         { "outSegs",          KSTAT_DATA_UINT64, 0 },
376     };
377
378     ksp = kstat_create_netstack(DCCP_MOD_NAME, 0, DCCP_MOD_NAME, "mib2",
379                                KSTAT_TYPE_NAMED, NUM_OF_FIELDS(dccp_named_kstat_t), 0, stackid);
380     if (ksp == NULL) {
381         return (NULL);
382     }
383
384     bcopy(&template, ksp->ks_data, sizeof (template));
385     ksp->ks_update = dccp_kstat_update;
386     ksp->ks_private = (void *) (uintptr_t) stackid;
387
388     kstat_install(ksp);
389
390     return (ksp);
391 }

```

```

393 /*
394  * Destroy DCCP kernel statistics.
395  */
396 void
397 dccp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
398 {
400     if (ksp != NULL) {
401         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
402         kstat_delete_netstack(ksp, stackid);
403     }
404 }

406 /*
407  * Update DCCP kernel statistics.
408  */
409 static int
410 dccp_kstat_update(kstat_t *kp, int rw)
411 {
412     conn_t          *connp;
413     connf_t         *connfp;
414     dccp_named_kstat_t *dccpkp;
415     dccp_t          *dccp;
416     dccp_stack_t    *dccps;
417     ip_stack_t      *ipst;
418     netstack_t      *ns;
419     netstackid_t    stackid;
420     mib2_dccp_t     dccp_mib;

422     if (rw == KSTAT_WRITE) {
423         return (EACCES);
424     }

426     stackid = (netstackid_t)(uintptr_t)kp->ks_private;
427     ns = netstack_find_by_stackid(stackid);
428     if (ns == NULL) {
429         return (-1);
430     }

432     dccps = ns->netstack_dccp;
433     if (dccps == NULL) {
434         netstack_rele(ns);
435         return (-1);
436     }

438     dccpkp = (dccp_named_kstat_t *)kp->ks_data;
439     ipst = ns->netstack_ip;

441     bzero(&dccp_mib, sizeof (dccp_mib));
442     dccp_sum_mib(dccps, &dccp_mib);

444     /* Fixed length structure for IPv4 and IPv6 counters */
445     SET_MIB(dccp_mib.dccpConnTableSize, sizeof (mib2_dccpConnEntry_t));
446     SET_MIB(dccp_mib.dccp6ConnTableSize, sizeof (mib2_dccp6ConnEntry_t));

448     dccpkp->activeOpens.value.ui32 = dccp_mib.dccpActiveOpens;
449     dccpkp->passiveOpens.value.ui32 = dccp_mib.dccpPassiveOpens;
450     dccpkp->inSegs.value.ui64 = dccp_mib.dccpHCInSegs;
451     dccpkp->outSegs.value.ui64 = dccp_mib.dccpHCOutSegs;

453     return (0);
454 }

456 /*
457  */

```

```

458 /*
459 void *
460 dccp_kstat2_init(netstackid_t stackid)
461 {
462     kstat_t *ksp;

464     dccp_stat_t template = {
465         { "dccp_sock_fallback",          KSTAT_DATA_UINT64, 0 },
466     };

468     ksp = kstat_create_netstack(DCCP_MOD_NAME, 0, "dccpstat", "net",
469         KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t), 0,
470         stackid);
471     if (ksp == NULL) {
472         return (NULL);
473     }

475     bcopy(&template, ksp->ks_data, sizeof (template));
476     ksp->ks_private = (void *) (uintptr_t)stackid;
477     ksp->ks_update = dccp_kstat2_update;

479     kstat_install(ksp);

481     return (ksp);
482 }

484 /*
485  * Destroy DCCP kernel statistics.
486  */
487 void
488 dccp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
489 {
490     if (ksp != NULL) {
491         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
492         kstat_delete_netstack(ksp, stackid);
493     }
494 }

496 /*
497  * Update routine for .
498  */
499 static int
500 dccp_kstat2_update(kstat_t *kp, int rw)
501 {
502     dccp_stack_t    *dccps;
503     dccp_stat_t     *stats;
504     netstack_t      *ns;
505     netstackid_t    stackid;
506     int             i;
507     int             cnt;

509     if (rw == KSTAT_WRITE) {
510         return (EACCES);
511     }

513     stackid = (netstackid_t)(uintptr_t)kp->ks_private;
514     ns = netstack_find_by_stackid(stackid);
515     if (ns == NULL) {
516         return (-1);
517     }

519     dccps = ns->netstack_dccp;
520     if (dccps == NULL) {
521         netstack_rele(ns);
522         return (-1);
523     }

```

```
525     stats = (dccp_stat_t *)kp->ks_data;
526     dccp_clr_stats(stats);

528     /* Sum up all stats */
529     cnt = dccps->dccps_sc_cnt;
530     for (i = 0; i < cnt; i++) {
531         dccp_add_stats(&dccps->dccps_sc[i]->dccp_sc_stats, stats);
532     }

534     netstack_rele(ns);

536     return (0);
537 }

539 /*
540  * Add stats from one to another.
541  */
542 static void
543 dccp_add_mib(mib2_dccp_t *from, mib2_dccp_t *to)
544 {
545     to->dccpActiveOpens += from->dccpActiveOpens;
546     to->dccpPassiveOpens += from->dccpPassiveOpens;
547     to->dccpInSegs += from->dccpInSegs;
548     to->dccpOutSegs += from->dccpOutSegs;
549 }

551 /*
552  * Sum up all MIB-II stats for a dccp_stack_t from all per CPU stats.
553  */
554 static void
555 dccp_sum_mib(dccp_stack_t *dccps, mib2_dccp_t *dccp_mib)
556 {
557     int    i;
558     int    cnt;

560     cnt = dccps->dccps_sc_cnt;
561     for (i = 0; i < cnt; i++) {
562         dccp_add_mib(&dccps->dccps_sc[i]->dccp_sc_mib, dccp_mib);
563     }
564 }

566 /*
567  * Set all dccp_stat_t counters to zero.
568  */
569 static void
570 dccp_clr_stats(dccp_stat_t *stats)
571 {
572     stats->dccp_sock_fallback.value.ui64 = 0;
573 }

575 /*
576  * Add counters from the per CPU stats.
577  */
578 static void
579 dccp_add_stats(dccp_stat_counter_t *from, dccp_stat_t *to)
580 {
581     to->dccp_sock_fallback.value.ui64 +=
582         from->dccp_sock_fallback;
583 }
584 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_timers.c

1

```
*****
7387 Sat Aug 18 10:37:11 2012
new/usr/src/uts/common/inet/dccp/dccp_timers.c
dccp: bring in timers
*****

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 setting and deleting timers.
33 */

35 #include <sys/types.h>
36 #include <sys/strlog.h>
37 #include <sys/strsun.h>
38 #include <sys/strsubr.h>
39 #include <sys/squeue_impl.h>
40 #include <sys/squeue.h>
41 #include <sys/callo.h>

43 #include <inet/common.h>
44 #include <inet/ip.h>
45 #include <inet/ip_ire.h>
46 #include <inet/ip_rts.h>
47 #include <inet/dccp_impl.h>

49 #include <sys/cmn_err.h>

51 kmem_cache_t *dccp_timercache;

53 static void dccp_timer_callback(void *);
54 static void dccp_timer_handler(void *, mblk_t *, void *, ip_rcv_attr_t *);
55 static void dccp_timer_free(dccp_t *, mblk_t *);

57 /*
58  * Allocate a new timer.
59 */
60 timeout_id_t
61 dccp_timeout(conn_t *connp, void (*f)(void *), hrtime_t tim)
```

new/usr/src/uts/common/inet/dccp/dccp_timers.c

2

```
62 {
63     dccp_t      *dccp = connp->conn_dccp;
64     dccp_timer_t *dccpt;
65     mblk_t      *mp;

67     cmn_err(CE_NOTE, "dccp_timers.c: dccp_timeout");

69     ASSERT(connp->conn_sqp != NULL);

71     if (dccp->dccp_timercache == NULL) {
72         mp = dccp_timermp_alloc(KM_NOSLEEP | KM_PANIC);
73     } else {
74         mp = dccp->dccp_timercache;
75         dccp->dccp_timercache = mp->b_next;
76         mp->b_next = NULL;
77         ASSERT(mp->b_wptr == NULL);
78     }

80     CONN_INC_REF(connp);
81     dccpt = (dccp_timer_t *)mp->b_rptr;
82     dccpt->connp = connp;
83     dccpt->dccpt_proc = f;
84     dccpt->dccpt_tid = timeout_generic(CALLOUT_NORMAL, dccp_timer_callback,
85         mp, tim * MICROSEC, CALLOUT_TCP_RESOLUTION, CALLOUT_FLAG_ROUNDUP);
86     /* CALLOUT_DCCP_RESOLUTION */
87     VERIFY(!(dccpt->dccpt_tid & CALLOUT_ID_FREE));

89     return ((timeout_id_t)mp);
90 }

92 /*
93  * Callback function.
94 */
95 static void
96 dccp_timer_callback(void *arg)
97 {
98     conn_t      *connp;
99     dccp_timer_t *dccpt;
100     mblk_t      *mp = (mblk_t *)arg;

102     dccpt = (dccp_timer_t *)mp->b_rptr;
103     connp = dccpt->connp;
104     SQUEUE_ENTER_ONE(connp->conn_sqp, mp, dccp_timer_handler, connp,
105         NULL, SQ_FILL, SQTAG_DCCP_TIMER);
106 }

108 /*
109  * Fires the timer callback function.
110 */
111 /* ARGSUSED */
112 static void
113 dccp_timer_handler(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
114 {
115     conn_t      *connp = (conn_t *)arg;
116     dccp_t      *dccp = connp->conn_dccp;
117     dccp_timer_t *dccpt;

119     cmn_err(CE_NOTE, "dccp_timers.c: dccp_timer_handler");

121     dccpt = (dccp_timer_t *)mp->b_rptr;

123     ASSERT(connp == dccpt->connp);
124     ASSERT((squeue_t *)arg2 == connp->conn_sqp);

126     if (dccpt->dccpt_tid & CALLOUT_ID_FREE) {
127         dccp_timer_free(connp->conn_dccp, mp);
```

```

128         return;
129     }

131     if (dccp->dccp_state != DCCPS_CLOSED) {
132         (*dccpt->dccpt_proc)(connp);
133     } else {
134         dccp->dccp_timer_tid = 0;
135     }

137     dccp_timer_free(connp->conn_dccp, mp);
138 }

140 clock_t
141 dccp_timeout_cancel(conn_t *connp, timeout_id_t id)
142 {
143     dccp_timer_t      *dccpt;
144     mblk_t             *mp = (mblk_t *)id;
145     clock_t            delta;

147     if (mp == NULL) {
148         return (-1);
149     }

151     dccpt = (dccp_timer_t *)mp->b_rptr;
152     ASSERT(dccpt->connp == connp);

154     delta = untimeout_default(dccpt->dccpt_tid, 0);

156     if (delta >= 0) {
157         dccp_timer_free(connp->conn_dccp, mp);
158         CONN_DEC_REF(connp);
159     } else {
160         dccpt->dccpt_tid |= CALLOUT_ID_FREE;
161         delta = 0;
162     }

164     return (TICK_TO_MSEC(delta));
165 }

167 /*
168  * Allocate per-dccp timer cache.
169  */
170 mblk_t *
171 dccp_timermp_alloc(int kmflags)
172 {
173     mblk_t *mp;

175     mp = (mblk_t *)kmem_cache_alloc(dccp_timer_cache, kmflags & ~KM_PANIC);
176     if (mp != NULL) {
177         mp->b_next = mp->b_prev = NULL;
178         mp->b_rptr = (uchar_t *)(&mp[1]);
179         mp->b_wptr = NULL;
180         mp->b_datap = NULL;
181         mp->b_queue = NULL;
182         mp->b_cont = NULL;
183     } else if (kmflags & KM_PANIC) {
184         /* XXX */
185     }

187     return (mp);
188 }

190 /*
191  * Free per-dccp timer cache.
192  */
193 void

```

```

194 dccp_timermp_free(dccp_t *dccp)
195 {
196     mblk_t *mp;

198     while ((mp = dccp->dccp_timer_cache) != NULL) {
199         ASSERT(mp->b_wptr == NULL);
200         dccp->dccp_timer_cache = dccp->dccp_timer_cache->b_next;
201         kmem_cache_free(dccp_timer_cache, mp);
202     }
203 }

205 /*
206  * Free timer event.
207  */
208 static void
209 dccp_timer_free(dccp_t *dccp, mblk_t *mp)
210 {
211     mblk_t *mpl = dccp->dccp_timer_cache;

213     if (mp->b_wptr != NULL) {
214         if (mp->b_wptr != (uchar_t *)-1) {
215             freeb(mp);
216         } else {
217             kmem_free(mp, (size_t)mp->b_datap);
218         }
219     } else if (mpl == NULL || mpl->b_next == NULL) {
220         mp->b_rptr = (uchar_t *)(&mp[1]);
221         mp->b_next = mpl;
222         dccp->dccp_timer_cache = mp;
223     } else {
224         kmem_cache_free(dccp_timer_cache, mp);
225     }
226 }

228 /*
229  * Stop all DCCP timers.
230  */
231 void
232 dccp_timers_stop(dccp_t *dccp)
233 {
234     if (dccp->dccp_timer_tid != 0) {
235         (void) DCCP_TIMER_CANCEL(dccp, dccp->dccp_timer_tid);
236         dccp->dccp_timer_tid = 0;
237     }

239     if (dccp->dccp_ka_tid != 0) {
240         (void) DCCP_TIMER_CANCEL(dccp, dccp->dccp_ka_tid);
241         dccp->dccp_ka_tid = 0;
242     }
243 }

245 /*
246  * Keepalive timer.
247  */
248 void
249 dccp_keepalive_timer(void *arg)
250 {
251     conn_t      *connp = (conn_t *)arg;
252     dccp_t      *dccp = connp->conn_dccp;
253     dccp_stack_t *dccps = dccp->dccp_dccps;
254     int32_t      firetime;
255     int32_t      idletime;
256     int32_t      ka_intrvl;

258     cmn_err(CE_NOTE, "dccp_timers.c: dccp_keepalive_timer");

```

```

260     dccp->dccp_ka_tid = 0;

262     DCCPS_BUMP_MIB(dccps, dccpTimKeepalive);
263     ka_intrvl = dccp->dccp_ka_interval;

265     if (dccp->dccp_state > DCCPS_CLOSING) {
266         return;
267     }

269     if (dccp->dccp_state < DCCPS_OPEN) {
270         dccp->dccp_ka_tid = DCCP_TIMER(dccp, dccp_keepalive_timer,
271             ka_intrvl);
272         return;
273     }

275     idletime = TICK_TO_MSEC(ddi_get_lbolt() - dccp->dccp_last_rcv_time);

277     /* XXX */

279     if ((firetime = ka_intrvl - idletime) < 0) {
280         firetime = ka_intrvl;
281     }

283     dccp->dccp_ka_tid = DCCP_TIMER(dccp, dccp_keepalive_timer, firetime);
284 }

286 /*
287  * Timer service routine.
288  */
289 void
290 dccp_timer(void *arg)
291 {
292     conn_t      *connp = (conn_t *)arg;
293     dccp_t      *dccp = connp->conn_dccp;
294     dccp_stack_t *dccps = dccp->dccp_dccps;
295     mblk_t      *mp;
296     clock_t      first_threshold;
297     clock_t      second_threshold;
298     clock_t      ms;
299     uint32_t      mss;
300     boolean_t     dont_timeout = B_FALSE;

302     cmn_err(CE_NOTE, "dccp_timers.c: dccp_timer");

304     dccp->dccp_timer_tid = 0;

306     first_threshold = dccp->dccp_first_timer_threshold;
307     second_threshold = dccp->dccp_second_timer_threshold;
308     switch (dccp->dccp_state) {
309     case DCCPS_REQUEST:
310         first_threshold = dccp->dccp_first_ctimer_threshold;
311         second_threshold = dccp->dccp_second_ctimer_threshold;

313         if (second_threshold == 0) {
314             second_threshold = dccps->dccps_ip_abort_linterval;
315             if (dccp->dccp_active_open) {
316                 dont_timeout = B_TRUE;
317             }
318         }
319         break;
320     case DCCPS_CLOSING:
321         (void) dccp_clean_death(dccp, 0);
322         return;
323     default:
324         if (connp->conn_debug) {
325             (void) strlog(DCCP_MOD_ID, 0, 1, SL_TRACE|SL_ERROR,

```

```

326         "dccp_timer: strange state (%d) %s",
327         dccp->dccp_state, dccp_display(dccp, NULL,
328             DISP_PORT_ONLY));
329     }
330     return;
331 }

333     ASSERT(second_threshold != 0);

335     ms = 900;

337     DCCP_TIMER_RESTART(dccp, ms);

339     DCCPS_BUMP_MIB(dccps, dccpRetransSegs);
340 }
341 #endif /* ! codereview */

```


new/usr/src/uts/common/inet/dccp/dccp_tpi.c

1

9481 Sat Aug 18 10:37:11 2012
new/usr/src/uts/common/inet/dccp/dccp_tpi.c
dccp: clean up

```
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  * Functions related to TPI.
24  */
25
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>
39
40 #include <inet/common.h>
41 #include <inet/dccp_impl.h>
42 #include <inet/ip.h>
43 #include <inet/proto_set.h>
44
45 #include <sys/cmn_err.h>
46
47 /*
48  * This file contains functions related to the TPI interface.
49  */
50
51 /*
52  * XXX
53  */
54
55 static void
56 dccp_copy_info(struct T_info_ack *tia, dccp_t *dccp)
57 {
58     conn_t      *connp = dccp->dccp_connp;
59     dccp_stack_t *dccps = dccp->dccp_dccps;
60     extern struct T_info_ack dccp_g_t_info_ack;
61     extern struct T_info_ack dccp_g_t_info_ack_v6;
```

new/usr/src/uts/common/inet/dccp/dccp_tpi.c

2

```
63     if (connp->conn_family == AF_INET6) {
64         *tia = dccp_g_t_info_ack_v6;
65     } else {
66         *tia = dccp_g_t_info_ack;
67     }
68
69     /* XXX */
70 }
71
72 /*
73  * XXX
74  */
75 void
76 dccp_do_capability_ack(dccp_t *dccp, struct T_capability_ack *tcap,
77     t_uscalar_t cap_bits1)
78 {
79     tcap->CAP_bits1 = 0;
80
81     if (cap_bits1 & TC1_INFO) {
82         dccp_copy_info(&tcap->INFO_ack, dccp);
83         tcap->CAP_bits1 |= TC1_INFO;
84     }
85
86     if (cap_bits1 & TC1_ACCEPTOR_ID) {
87         tcap->ACCEPTOR_id = dccp->dccp_acceptor_id;
88         tcap->CAP_bits1 |= TC1_ACCEPTOR_ID;
89     }
90 }
91
92 /*
93  * This routine responds to T_CAPABILITY_REQ messages.
94  */
95 void
96 dccp_capability_req(dccp_t *dccp, mblk_t *mp)
97 {
98     struct T_capability_ack *tcap;
99     t_uscalar_t cap_bits1;
100
101     if (MBLKL(mp) < sizeof (struct T_capability_req)) {
102         freemsg(mp);
103         return;
104     }
105
106     cap_bits1 = ((struct T_capability_req *)mp->b_rptr->CAP_bits1;
107
108     mp = tpi_ack_alloc(mp, sizeof (struct T_capability_ack),
109         mp->b_datap->db_type, T_CAPABILITY_ACK);
110     if (mp == NULL) {
111         return;
112     }
113
114     tcap = (struct T_capability_ack *)mp->b_rptr;
115     dccp_do_capability_ack(dccp, tcap, cap_bits1);
116
117     putnext(dccp->dccp_connp->conn_rq, mp);
118 }
119
120 /*
121  * This routine responds to T_INFO_REQ messages.
122  */
123 void
124 dccp_info_req(dccp_t *dccp, mblk_t *mp)
125 {
126     mp = tpi_ack_alloc(mp, sizeof (struct T_info_ack), M_PCPROTO,
127         T_INFO_ACK);
```

```

128     if (mp == NULL) {
129         dccp_err_ack(dccp, mp, TSYSEERR, ENOMEM);
130         return;
131     }

133     dccp_copy_info((struct T_info_ack *)mp->b_rptr, dccp);
134     putnext(dccp->dccp_connp->conn_rq, mp);
135 }

137 /*
138  * This routine responds to T_ADDR_REQ messages.
139  */
140 void
141 dccp_addr_req(dccp_t *dccp, mblk_t *mp)
142 {
143     conn_t *connp = dccp->dccp_connp;
144     mblk_t *ackmp;
145     uint_t addrlen;
146     struct sockaddr *sa;
147     struct T_addr_ack *taa;

149     /* XXX */
150 }

152 /*
153  * Helper function to generate TPI errors acks.
154  */
155 void
156 dccp_err_ack(dccp_t *dccp, mblk_t *mp, int t_error, int sys_error)
157 {
158     if ((mp = mi_tpi_err_ack_alloc(mp, t_error, sys_error)) != NULL) {
159         putnext(dccp->dccp_connp->conn_rq, mp);
160     }
161 }

163 void
164 dccp_tpi_connect(dccp_t *dccp, mblk_t *mp)
165 {
166     conn_t *connp = dccp->dccp_connp;
167     queue_t *q = connp->conn_wq;
168     struct sockaddr *sa;
169     struct T_conn_req *tcr;
170     sin_t *sin;
171     sin6_t *sin6;
172     cred_t *cr;
173     pid_t cpid;
174     socklen_t len;
175     int error;

177     cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_connect");

179     cr = msg_getcred(mp, &cpid);
180     ASSERT(cr != NULL);
181     if (cr == NULL) {
182         dccp_err_ack(dccp, mp, TSYSEERR, EINVAL);
183         return;
184     }

186     tcr = (struct T_conn_req *)mp->b_rptr;

188     ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <= (uintptr_t)INT_MAX);
189     if ((mp->b_wptr - mp->b_rptr) < sizeof (*tcr)) {
190         dccp_err_ack(dccp, mp, TPROTO, 0);
191         return;
192     }

```

```

194     error = proto_verify_ip_addr(connp->conn_family, sa, len);
195     if (error != 0) {
196         dccp_err_ack(dccp, mp, TSYSEERR, 0);
197         return;
198     }

200     error = dccp_do_connect(dccp->dccp_connp, sa, len, cr, cpid);
201     if (error < 0) {
202         mp = mi_tpi_err_ack_alloc(mp, -error, 0);
203     } else if (error > 0) {
204         mp = mi_tpi_err_ack_alloc(mp, TSYSEERR, error);
205     } else {
206         mp = mi_tpi_ok_ack_alloc(mp);
207     }
208 }

210 int
211 dccp_tpi_close(queue_t *q, int flags)
212 {
213     conn_t *connp;

215     ASSERT(WR(q)->q_next == NULL);

217     connp = Q_TO_CONN(q);

219     dccp_close_common(connp, flags);

221     qprocsoff(q);
222     inet_minor_free(connp->conn_minor_arena, connp->conn_dev);

224     return (0);
225 }

227 int
228 dccp_tpi_close_accept(queue_t *q)
229 {
230     vmem_t *minor_arena;
231     dev_t conn_dev;

233     cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_close_accept");

235     return (0);
236 }

238 boolean_t
239 dccp_conn_con(dccp_t *dccp, uchar_t *iphdr, mblk_t *idmp,
240     mblk_t **defermp, ip_recv_attr_t *ira)
241 {
242     conn_t *connp = dccp->dccp_connp;
243     sin_t sin;
244     sin6_t sin6;
245     mblk_t *mp;
246     char *optp = NULL;
247     int optlen = 0;

249     cmn_err(CE_NOTE, "dccp_tpi.c: dccp_conn_con");

251     if (defermp != NULL) {
252         *defermp = NULL;
253     }

255     if (IPH_HDR_VERSION(iphdr) == IPV4_VERSION) {
256         /* Packet is IPv4 */
257         if (connp->conn_family == AF_INET) {
258             sin = sin_null;
259             sin.sin_addr.s_addr = connp->conn_faddr_v4;

```

```

260         sin.sin_port = connp->conn_fport;
261         sin.sin_family = AF_INET;
262         mp = mi_tpi_conn_con(NULL, (char *)&sin,
263             (int)sizeof(sin_t), optp, optlen);
264     } else {
265         sin6 = sin6_null;
266         sin6.sin6_addr = connp->conn_faddr_v6;
267         sin6.sin6_port = connp->conn_fport;
268         sin6.sin6_family = AF_INET6;
269         mp = mi_tpi_conn_con(NULL, (char *)&sin6,
270             (int)sizeof(sin6_t), optp, optlen);
271     }
272 } else {
273     ip6_t    *ip6h = (ip6_t *)iphdr;
274
275     ASSERT(IPH_HDR_VERSION(iphdr) == IPV6_VERSION);
276     ASSERT(connp->conn_family == AF_INET6);
277
278     sin6 = sin6_null;
279     sin6.sin6_addr = connp->conn_faddr_v6;
280     sin6.sin6_port = connp->conn_fport;
281     sin6.sin6_family = AF_INET6;
282     sin6.sin6_flowinfo = ip6h->ip6_vcf & ~IPV6_VERS_AND_FLOW_MASK;
283     mp = mi_tpi_conn_con(NULL, (char *)&sin6,
284         (int)sizeof(sin6_t), optp, optlen);
285 }
286
287 if (!mp) {
288     return (B_FALSE);
289 }
290
291 mblk_copycrred(mp, idmp);
292
293 if (defermp == NULL) {
294     if (IPCL_IS_NONSTR(connp)) {
295         cmn_err(CE_NOTE, "calling su_connected");
296         (*connp->conn_upcalls->su_connected)
297             (connp->conn_upper_handle, dccp->dccp_connid,
298             ira->ira_cred, ira->ira_cpid);
299         freemsg(mp);
300     } else {
301         if (ira->ira_cred != NULL) {
302             /* So that getpeerucred works for TPI sockfs */
303             mblk_setcred(mp, ira->ira_cred, ira->ira_cpid);
304         }
305         putnext(connp->conn_rq, mp);
306     }
307 } else {
308     *defermp = mp;
309 }
310
311 /* XXX */
312 return (B_TRUE);
313 }
314
315 /*
316  * Routine to get the values of options.
317  */
318 int
319 dccp_tpi_opt_get(queue_t *q, int level, int name, uchar_t *ptr)
320 {
321     return (dccp_opt_get(Q_TO_CONN(q), level, name, ptr));
322 }
323
324 /*
325  * Routine to set the values of options.

```

```

326  */
327  /* ARGSUSED */
328  int
329  dccp_tpi_opt_set(queue_t *q, uint_t optset_context, int level, int name,
330      uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
331      void *thisdg_attrs, cred_t *cr)
332  {
333      conn_t    *connp = Q_TO_CONN(q);
334
335      return (dccp_opt_set(connp, optset_context, level, name, inlen, invalp,
336          outlenp, outvalp, thisdg_attrs, cr));
337  }
338
339  void
340  dccp_tpi_accept(queue_t *q, mblk_t *mp)
341  {
342      struct T_ok_ack    *ok;
343      struct T_conn_res  *conn_res;
344      conn_t              *econnp;
345      dccp_t              *eager;
346      dccp_t              *listener;
347      queue_t             *rq = RD(q);
348      mblk_t              *discon_mp;
349      cred_t              *cr;
350
351      cmn_err(CE_NOTE, "dccp_tpi.c: dccp_tpi_accept");
352
353      ASSERT(DB_TYPE(mp) == M_PROTO);
354
355      cr = msg_getcred(mp, NULL);
356      ASSERT(cr != NULL);
357      if (cr == NULL) {
358          mp = mi_tpi_err_ack_alloc(mp, TSYSERR, EINVAL);
359          if (mp != NULL) {
360              putnext(rq, mp);
361          }
362          return;
363      }
364
365      conn_res = (struct T_conn_res *)mp->b_rptr;
366      ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <= (uintptr_t)INT_MAX);
367      if ((mp->b_wptr - mp->b_rptr) < sizeof(struct T_conn_res)) {
368          mp = mi_tpi_err_ack_alloc(mp, TPROTO, 0);
369          if (mp != NULL) {
370              putnext(rq, mp);
371          }
372          return;
373      }
374
375      switch (conn_res->PRIM_type) {
376      case O_T_CONN_RES:
377      case T_CONN_RES:
378          putnext(rq, mp);
379          return;
380      default:
381          mp = mi_tpi_err_ack_alloc(mp, TNOTSUPPORT, 0);
382          if (mp != NULL) {
383              putnext(rq, mp);
384          }
385          return;
386      }
387  }
388
389  static void
390  dccp_accept_swap(dccp_t *listener, dccp_t *acceptor, dccp_t *eager)
391  {

```

```
392     conn_t *econnp;
393     conn_t *aconnp;
394 }

396 /*
397  * This runs at the tail end of accept processing on the queue of the
398  * new connection.
399  */
400 static void
401 dccp_accept_finish(void *arg, mblk_t *mp, void *arg2, ip_rcv_attr_t *dummy)
402 {
403     conn_t      *connp = (conn_t *)arg;
404     dccp_t      *dccp = connp->conn_dccp;
405     dccp_stack_t *dccps = dccp->dccp_dccps;
406     queue_t      *q = connp->conn_rq;
407     struct stroptions *stropt;
408     struct sock_proto_props sopp;

410     /* Should never be called for non-STREAMS sockets */
411     ASSERT(!IPCL_IS_NONSTR(connp));

413     /* We should just receive a single mblk that fits a T_discon_ind */
414     ASSERT(mp->b_cont == NULL);

416     /* XXX */

418     dccp_get_proto_props(dccp, &sopp);

420     ASSERT(DB_REF(mp) == 1);
421     ASSERT(MBLKSIZE(mp) >= sizeof (struct stroptions));

423     DB_TYPE(mp) = M_SETOPTS;

425     /* XXX */

427     dccp->dccp_hard_binding = B_FALSE;

429     if (connp->conn_keepalive) {
430         dccp->dccp_ka_last_intrvl = 0;
431         dccp->dccp_ka_tid = DCCP_TIMER(dccp, dccp_keepalive_timer,
432             dccp->dccp_ka_interval);
433     }
434 }
435 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccp_tunables.c

1

2583 Sat Aug 18 10:37:11 2012
new/usr/src/uts/common/inet/dccp/dccp_tunables.c
dccp: starting module template

```
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 Hoeppner. 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_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} },
29
30     { "smallest_anon_port", MOD_PROTO_DCCP,
31       mod_set_uint32, mod_get_uint32,
32       {1024, ULP_MAX_PORT, 32*1024}, {32*1024} },
33
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} },
37
38     { "_xmit_lowat", MOD_PROTO_DCCP,
39       mod_set_uint32, mod_get_uint32,
40       {0, (1<<30), DCCP_XMIT_LOWATER},
41       {DCCP_XMIT_LOWATER} },
42
43     { "_debug", MOD_PROTO_DCCP,
44       mod_set_uint32, mod_get_uint32,
45       {0, 10, 0}, {0} },
46
47     { "_rst_sent_rate_enabled", MOD_PROTO_DCCP,
48       mod_set_boolean, mod_get_boolean,
49       {B_TRUE}, {B_TRUE} },
50
51     { "_rst_sent_rate", MOD_PROTO_DCCP,
52       mod_set_uint32, mod_get_uint32,
53       {0, UINT32_MAX, 40}, {40} },
54
55     { "_ip_abort_cinterval", MOD_PROTO_DCCP,
56       mod_set_uint32, mod_get_uint32,
57       {1*SECONDS, UINT32_MAX, 3*MINUTES}, {3*MINUTES} },
58
59     { "_ip_abort_linterval", MOD_PROTO_DCCP,
60       mod_set_uint32, mod_get_uint32,
61       {1*SECONDS, UINT32_MAX, 3*MINUTES}, {3*MINUTES} },
```

new/usr/src/uts/common/inet/dccp/dccp_tunables.c

2

```
63     { "_ip_abort_interval", MOD_PROTO_DCCP,
64       mod_set_uint32, mod_get_uint32,
65       {500*MS, UINT32_MAX, 5*MINUTES}, {5*MINUTES} },
66
67     /* tunable - 10 */
68     { "_ip_notify_cinterval", MOD_PROTO_DCCP,
69       mod_set_uint32, mod_get_uint32,
70       {1*SECONDS, UINT32_MAX, 10*SECONDS}, {10*SECONDS} },
71
72     { "_ip_notify_interval", MOD_PROTO_DCCP,
73       mod_set_uint32, mod_get_uint32,
74       {500*MS, UINT32_MAX, 10*SECONDS}, {10*SECONDS} },
75
76     { "_keepalive_interval", MOD_PROTO_DCCP,
77       mod_set_uint32, mod_get_uint32,
78       {10*SECONDS, 10*HOURS, 2*HOURS}, {2*HOURS} },
79
80     { "_keepalive_abort_interval", MOD_PROTO_DCCP,
81       mod_set_uint32, mod_get_uint32,
82       {0, UINT32_MAX, 8*MINUTES}, {8*MINUTES} },
83
84     { NULL, 0, NULL, NULL, {0}, {0} }
85 };
86
87 int dccp_propinfo_count = A_CNT(dccp_propinfo_tbl);
88 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp/dccpddi.c

1

1466 Sat Aug 18 10:37:11 2012

new/usr/src/uts/common/inet/dccp/dccpddi.c

dccp: starting module template

```
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 Hoeppner. 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>
23 #include <inet/dccp.h>
24 #include <inet/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 *modinfo)
57 {
58     return (mod_info(&modlinkage, modinfo));
59 }
60 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp_impl.h

1

11716 Sat Aug 18 10:37:11 2012

new/usr/src/uts/common/inet/dccp_impl.h

***ccp: move headers into inet dir, like other protocols

```
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 Hoeppner. 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>
23 #include <sys/clock_impl.h>

25 #include <netinet/in.h>
26 #include <netinet/ip6.h>
27 #include <netinet/dccp.h>
```

```
29 #include <inet/common.h>
30 #include <inet/dccp.h>
31 #include <inet/ip.h>
32 #include <inet/ip6.h>
33 #include <inet/optcom.h>
34 #include <inet/tunables.h>
```

```
36 #include <inet/dccp_stack.h>
```

```
38 #ifdef __cplusplus
39 extern "C" {
40 #endif
```

```
42 #ifdef _KERNEL
```

```
44 #define DCCP_MOD_ID          5999 /* XXX */
```

```
46 extern struct qinit      dccp_sock_winit;
47 extern struct qinit      dccp_winit;
```

```
49 extern sock_downcalls_t sock_dccp_downcalls;
```

```
51 #define DCCP_XMIT_LOWATER      (4 * 1024)
52 #define DCCP_XMIT_HIWATER      49152
53 #define DCCP_RECV_LOWATER      (2 * 1024)
54 #define DCCP_RECV_HIWATER      128000
```

```
56 #define TIDUSZ 4096 /* transport interface data unit size */
```

```
58 /*
59  * Format arguments passed to dccp_display.
60 */
61 #define DISP_PORT_ONLY        1
```

new/usr/src/uts/common/inet/dccp_impl.h

2

```
62 #define DISP_ADDR_AND_PORT    2
```

```
64 /*
65  * Bind hash array size and hash function.
66 */
```

```
67 #define DCCP_BIND_FANOUT_SIZE    128
68 #define DCCP_BIND_HASH(lport, size) ((ntohs((uint16_t)lport)) & (size - 1))
```

```
70 /*
71  * Was this tcp created via socket() interface?
72 */
73 #define DCCP_IS_SOCKET(dccp)    ((dccp)->dccp_issocket)
```

```
75 /*
76  * Is this dccp not attached to any upper client?
77 */
78 #define DCCP_IS_DETACHED(dccp) ((dccp)->dccp_detached)
```

```
80 /* Packet types (RFC 4340, Section 5.1.) */
```

```
81 #define DCCP_PKT_REQUEST        0
82 #define DCCP_PKT_RESPONSE       1
83 #define DCCP_PKT_DATA           2
84 #define DCCP_PKT_ACK            3
85 #define DCCP_PKT_DATAACK        4
86 #define DCCP_PKT_CLOSEREQ       5
87 #define DCCP_PKT_CLOSE          6
88 #define DCCP_PKT_RESET          7
89 #define DCCP_PKT_SYNC           8
90 #define DCCP_PKT_SYNCACK        9
```

```
92 #define DCCP_PACKET_REQUEST     0
93 #define DCCP_PACKET_RESPONSE    1
94 #define DCCP_PACKET_DATA        2
95 #define DCCP_PACKET_ACK         3
96 #define DCCP_PACKET_DATAACK     4
97 #define DCCP_PACKET_CLOSEREQ    5
98 #define DCCP_PACKET_CLOSE       6
99 #define DCCP_PACKET_RESET       7
100 #define DCCP_PACKET_SYNC        8
101 #define DCCP_PACKET_SYNCACK     9
```

```
103 /*
104  * DCCP options and features.
105 */
```

```
107 /*
108  * Options types (RFC 4340, Section 5.8.)
109 */
```

```
110 #define DCCP_OPTION_PADDING      0
111 #define DCCP_OPTION_MANDATORY    1
112 #define DCCP_OPTION_SLOW_RECEIVER 2
113 #define DCCP_OPTION_CHANGE_L    32
114 #define DCCP_OPTION_CONFIRM_L   33
115 #define DCCP_OPTION_CHANGE_R    34
116 #define DCCP_OPTION_CONFIRM_R   35
117 #define DCCP_OPTION_INIT_COOKIE 36
118 #define DCCP_OPTION_NDP_COUNT    37
119 #define DCCP_OPTION_ACK_VECTOR_1 38
120 #define DCCP_OPTION_ACK_VECTOR_2 39
121 #define DCCP_OPTION_DATA_DROPPED 40
122 #define DCCP_OPTION_TIMESTAMP    41
123 #define DCCP_OPTION_TIMESTAMP_ECHO 42
124 #define DCCP_OPTION_ELAPSED_TIME 43
125 #define DCCP_OPTION_DATA_CHECKSUM 44
```

```
127 /*
```

```

128 * Feature types (RFC 4340, Section 6.4.)
129 */
130 #define DCCP_FEATURE_CCID 1
131 #define DCCP_FEATURE_ALLOW_SHORT_SEQNOS 2
132 #define DCCP_FEATURE_SEQUENCE_WINDOW 3
133 #define DCCP_FEATURE_ECN_INCAPABLE 4
134 #define DCCP_FEATURE_ACK_RATIO 5
135 #define DCCP_FEATURE_SEND_ACK_VECTOR 6
136 #define DCCP_FEATURE_SEND_NDP_COUNT 7
137 #define DCCP_FEATURE_MIN_CHECKSUM_COVERAGE 8
138 #define DCCP_FEATURE_CHECK_DATA_CHECKSUM 9

140 /*
141 * Feature negotiation states (RFC 4340, Section 6.6.2.)
142 */
143 #define DCCP_FEATURE_STATE_CHANGING 0
144 #define DCCP_FEATURE_STATE_UNSTABLE 1
145 #define DCCP_FEATURE_STATE_STABLE 2

147 /*
148 * Reset types (RFC 4230, Section 5.6.)
149 */
150 #define DCCP_RESET_UNSPECIFIED 0
151 #define DCCP_RESET_CLOSED 1
152 #define DCCP_RESET_ABORTED 2
153 #define DCCP_RESET_NO_CONNECTION 3
154 #define DCCP_RESET_PACKET_ERROR 4
155 #define DCCP_RESET_OPTION_ERROR 5
156 #define DCCP_RESET_MANDATORY_ERROR 6
157 #define DCCP_RESET_CONNECTION_REFUSED 7
158 #define DCCP_RESET_BAD_SERVICE_CODE 8
159 #define DCCP_RESET_TOO_BUSY 9
160 #define DCCP_RESET_BAD_INIT_COOKIE 10
161 #define DCCP_RESET_AGGRESSION_PENALTY 11
162 #define DCCP_RESET_RESERVED 12

164 typedef struct dccp_feature_s {
165     list_node_t df_next;
166     uint8_t df_option;
167     uint8_t df_type;
168     uint8_t df_state;
169     uint64_t df_value;
170     boolean_t df_mandatory;
171 } dccp_feature_t;

173 /* Options in DCCP header */
174 typedef struct dccp_opt_s {
175     int type;
176     boolean_t mandatory;
177 } dccp_opt_t;

179 /*
180 * Tuneables.
181 */
182 #define dccps_smallest_nonpriv_port dccps_propinfo_tbl[0].prop_cur_uval
183 #define dccps_smallest_anon_port dccps_propinfo_tbl[1].prop_cur_uval
184 #define dccps_largest_anon_port dccps_propinfo_tbl[2].prop_cur_uval

186 #define dccps_dbg dccps_propinfo_tbl[4].prop_cur_uval
187 #define dccps_rst_sent_rate_enabled dccps_propinfo_tbl[5].prop_cur_uval
188 #define dccps_rst_sent_rate dccps_propinfo_tbl[6].prop_cur_uval

190 #define dccps_ip_abort_cinterval dccps_propinfo_tbl[7].prop_cur_uval
191 #define dccps_ip_abort_linterval dccps_propinfo_tbl[8].prop_cur_uval
192 #define dccps_ip_abort_interval dccps_propinfo_tbl[9].prop_cur_uval
193 #define dccps_ip_notify_cinterval dccps_propinfo_tbl[10].prop_cur_uval

```

```

194 #define dccps_ip_notify_interval dccps_propinfo_tbl[11].prop_cur_uval
195 #define dccps_keepalive_interval_high dccps_propinfo_tbl[12].prop_max_uval
196 #define dccps_keepalive_interval dccps_propinfo_tbl[12].prop_cur_uval
197 #define dccps_keepalive_interval_low dccps_propinfo_tbl[12].prop_min_uval
198 #define dccps_keepalive_abort_interval_high \
199     dccps_propinfo_tbl[13].prop_max_uval
200 #define dccps_keepalive_abort_interval dccps_propinfo_tbl[13].prop_cur_uval
201 #define dccps_keepalive_abort_interval_low \
202     dccps_propinfo_tbl[13].prop_min_uval

204 /*
205 * Timers.
206 */
207 typedef struct dccp_timer_s {
208     conn_t *connp;
209     void (*dcept_proc)(void *);
210     callout_id_t dcept_tid;
211 } dccp_timer_t;

213 extern kmem_cache_t *dccp_timer_cache;

215 #define DCCP_TIMER(dccp, f, tim) \
216     dccp_timeout(dccp->dccp_connp, f, tim)

218 #define DCCP_TIMER_CANCEL(dccp, id) \
219     dccp_timeout_cancel(dccp->dccp_connp, id)

221 #define DCCP_TIMER_RESTART(dccp, intvl) { \
222     if ((dccp)->dccp_timer_tid != 0) \
223         (void) DCCP_TIMER_CANCEL(dccp, (dccp)->dccp_timer_tid); \
224     (dccp)->dccp_timer_tid = DCCP_TIMER(dccp, dccp_timer, (intvl)); \
225 }

227 extern struct qinit dccp_rinitv4, dccp_rinitv6;

229 extern optdb_obj_t dccp_opt_obj;
230 extern uint_t dccp_max_optsize;

232 extern int dccp_queue_flag;

234 /*
235 * Functions in dccp.c
236 */
237 extern int dccp_build_hdrs(dccp_t *);
238 extern conn_t *dccp_create_common(cred_t *, boolean_t, boolean_t, int *);
239 extern void dccp_close_common(conn_t *, int);
240 extern int dccp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
241     boolean_t);
242 extern int dccp_do_unbind(conn_t *);
243 extern int dccp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
244     cred_t *, boolean_t);
245 extern int dccp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
246     cred_t *, pid_t);
247 extern void dccp_init_values(dccp_t *, dccp_t *);
248 extern void dccp_free(dccp_t *);
249 extern void *dccp_get_conn(void *, dccp_stack_t *);
250 extern void dccp_get_proto_props(dccp_t *, struct sock_proto_props *);
251 extern int dccp_set_destination(dccp_t *);
252 extern int dccp_clean_death(dccp_t *, int);

255 /*
256 * Functions in dccp_bind.c
257 */
258 extern void dccp_bind_hash_insert(dccp_df_t *, dccp_t *, int);
259 extern void dccp_bind_hash_remove(dccp_t *);

```



```

260 extern int      dccp_bind_check(conn_t *, struct sockaddr *, socklen_t,
261                                cred_t *, boolean_t);
262 extern in_port_t dccp_bindi(dccp_t *, in_port_t, const in6_addr_t *, int,
263                             boolean_t, boolean_t);
264 extern in_port_t dccp_update_next_port(in_port_t, const dccp_t *, boolean_t);

266 /*
267  * Functions in dccp_features.c
268  */
269 extern int      dccp_parse_feature(dccp_t *, uint8_t, uint8_t, uchar_t *,
270                                   boolean_t);

272 /*
273  * Functions in dccp_stats.c
274  */
275 extern mblk_t   *dccp_snmp_get(queue_t *, mblk_t *, boolean_t);
276 extern void      *dccp_kstat_init(netstackid_t);
277 extern void      dccp_kstat_fini(netstackid_t, kstat_t *);
278 extern void      *dccp_kstat2_init(netstackid_t);
279 extern void      dccp_kstat2_fini(netstackid_t, kstat_t *);

281 /*
282  * Functions in dccp_socket.c
283  */
284 extern sock_lower_handle_t dccp_create(int, int, int, sock_downcalls_t **,
285                                       uint_t *, int *, int, cred_t *);
286 extern int      dccp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
287                               so_proto_quiesced_cb_t, sock_quiesce_arg_t *);
288 extern boolean_t dccp_newconn_notify(dccp_t *, ip_recv_attr_t *);

290 /*
291  * Functions in dccp_input.c
292  */
293 extern void      dccp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
294 extern void      dccp_input_data(void *, mblk_t *, void *, ip_recv_attr_t *);
295 extern void      dccp_rsrv(queue_t *);
296 extern void      dccp_input_listener_unbound(void *, mblk_t *, void *,
297                                              ip_recv_attr_t *);
298 extern boolean_t dccp_verifyicmp(conn_t *, void *, icmp_h_t *, icmp6_t *,
299                                 ip_recv_attr_t *);

301 /*
302  * Functions in dccp_misc.c
303  */
304 extern void      dccp_stack_cpu_add(dccp_stack_t *, processorid_t);
305 extern char      *dccp_display(dccp_t *, char *, char);

307 /*
308  * Functions in dccp_options.c
309  */
310 extern int      dccp_parse_options(dccp_t *, dccpha_t *);
311 extern int      dccp_process_options(dccp_t *, dccpha_t *);
312 extern int      dccp_generate_options(dccp_t *, void **, size_t *);

314 /*
315  * Functions in dccp_output.c
316  */
317 extern void      dccp_wput(queue_t *, mblk_t *);
318 extern void      dccp_wput_data(dccp_t *, mblk_t *, boolean_t);
319 extern void      dccp_wput_sock(queue_t *, mblk_t *);
320 extern void      dccp_wput_fallback(queue_t *, mblk_t *);
321 extern void      dccp_output(void *, mblk_t *, void *, ip_recv_attr_t *);
322 extern void      dccp_output_urgent(void *, mblk_t *, void *, ip_recv_attr_t *);
323 extern void      dccp_close_output(void *, mblk_t *, void *, ip_recv_attr_t *);
324 extern void      dccp_shutdown_output(void *, mblk_t *, void *, ip_recv_attr_t *);
325 extern void      dccp_send_data(dccp_t *, mblk_t *);

```

```

326 extern void      dccp_xmit_listeners_reset(mblk_t *, ip_recv_attr_t *,
327                                             ip_stack_t *, conn_t *);
328 extern void      dccp_send_synack(void *, mblk_t *, void *, ip_recv_attr_t *);
329 extern mblk_t     *dccp_xmit_mp(dccp_t *, mblk_t *, int32_t, int32_t *,
330                                 mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);
331 /* XXX following functions should be redone */
332 extern mblk_t     *dccp_generate_response(conn_t *, mblk_t *);
333 extern mblk_t     *dccp_generate_request(conn_t *);
334 extern mblk_t     *dccp_generate_reset(conn_t *);
335 extern mblk_t     *dccp_generate_ack(conn_t *);
336 extern mblk_t     *dccp_generate_sync(conn_t *);

338 /*
339  * Functions in dccp_opt_data.c
340  */
341 extern int      dccp_opt_get(conn_t *, int, int, uchar_t *);
342 extern int      dccp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
343                             uint_t *, uchar_t *, void *, cred_t *);

345 /*
346  * Functions in dccp_timers.c
347  */
348 extern timeout_id_t dccp_timeout(conn_t *, void (*)(void *), hrttime_t);
349 extern clock_t     dccp_timeout_cancel(conn_t *, timeout_id_t);
350 extern mblk_t     *dccp_timermp_alloc(int);
351 extern void      dccp_timermp_free(dccp_t *);
352 extern void      dccp_timers_stop(dccp_t *);
353 extern void      dccp_keepalive_timer(void *);
354 extern void      dccp_timer(void *);

356 /*
357  * Functions in dccp_tpi.c
358  */
359 extern void      dccp_do_capability_ack(dccp_t *, struct T_capability_ack *,
360                                         t_uscalar_t);
361 extern void      dccp_capability_req(dccp_t *, mblk_t *);
362 extern void      dccp_err_ack(dccp_t *, mblk_t *, int, int);
363 extern void      dccp_tpi_connect(dccp_t *, mblk_t *);
364 extern int      dccp_tpi_close(queue_t *, int);
365 extern int      dccp_tpi_close_accept(queue_t *);
366 extern boolean_t dccp_conn_con(dccp_t *, uchar_t *, mblk_t *, mblk_t **,
367                                ip_recv_attr_t *);
368 extern int      dccp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
369 extern int      dccp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
370                                  uint_t *, uchar_t *, void *, cred_t *);
371 extern void      dccp_tpi_accept(queue_t *, mblk_t *);

373 #endif /* _KERNEL */

375 #ifdef __cplusplus
376 }
377 #endif

379 #endif /* _INET_DCCP_IMPL_H */
380 #endif /* ! codereview */

```

new/usr/src/uts/common/inet/dccp_ip.h

1

802 Sat Aug 18 10:37:12 2012

new/usr/src/uts/common/inet/dccp_ip.h

dccp: move headers into inet dir, like other protocols

```
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 Hoeppner. 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_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_stack.h

1

```
*****
1624 Sat Aug 18 10:37:12 2012
new/usr/src/uts/common/inet/dccp_stack.h
dccp: move headers into inet dir, like other protocols
*****

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 Hoeppner. 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>
21 #include <sys/sunddi.h>
22 #include <sys/sunldi.h>
23 #include <inet/ip.h>
24 #include <inet/ipdrop.h>
25 #include <inet/dccp_stats.h>

27 #ifdef __cplusplus
28 extern "C" {
29 #endif

31 /*
32  * DCCP stack instances
33  */
34 typedef struct dccp_stack {
35     netstack_t          *dccps_netstack;      /* Common netstack */
36
37     uint_t               dccps_bind_fanout_size;
38     struct dccp_df_s     *dccps_bind_fanout;

40     /*
41      * Privileged ports.
42      */
43 #define DCCP_NUM_EPRIV_PORTS 64
44     int                  dccps_num_epriv_ports;
45     in_port_t            dccps_epriv_ports[DCCP_NUM_EPRIV_PORTS];
46     kmutex_t             dccps_epriv_port_lock;

48     in_port_t            dccps_min_anonpriv_port;
49     uint_t               dccps_next_port_to_try;

51     /* Reset rate control */
52     int64_t              dccps_last_rst_intrvl;
53     uint32_t             dccps_rst_cnt;

55     /* Tunables table */
56     struct mod_prop_info_s *dccps_propinfo_tbl;

58     ldi_ident_t          dccps_ldi_ident;

60     /* MIB-II kernel statistics */
61     kstat_t              *dccps_mibkp;
```

new/usr/src/uts/common/inet/dccp_stack.h

2

```
62     kstat_t              *dccps_kstat;

64     /* CPU stats counter */
65     dccp_stats_cpu_t     **dccps_sc;
66     int                  dccps_sc_cnt;
67 } dccp_stack_t;

69 #ifdef __cplusplus
70 }
71 #endif

73 #endif /* _INET_DCCP_DCCP_STACK_H */
74 #endif /* ! codereview */
```

new/usr/src/uts/common/inet/dccp_stats.h

1

1430 Sat Aug 18 10:37:12 2012
new/usr/src/uts/common/inet/dccp_stats.h
dccp: move headers into inet dir, like other protocols

```
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 Hoeppner. All rights reserved.
14 */

16 #ifndef _INET_DCCP_DCCP_STATS_H
17 #define _INET_DCCP_DCCP_STATS_H

19 #include <sys/netstack.h>
20 #include <sys/cpuvar.h>

22 #ifdef __cplusplus
23 extern "C" {
24 #endif

26 /* Kstats */
27 typedef struct dccp_stat {
28     kstat_named_t    dccp_sock_fallback;
29 } dccp_stat_t;

31 typedef struct dccp_stat_counter_s {
32     uint64_t          dccp_no_listener;
33     uint64_t          dccp_listendrop;
34     uint64_t          dccp_wsrsv_called;
35     uint64_t          dccp_sock_fallback;
36     uint64_t          dccp_rst_unsent;
37 } dccp_stat_counter_t;

39 typedef struct {
40     uint64_t          dccp_stats_cnt;
41     mib2_dccp_t       dccp_sc_mib;
42     dccp_stat_counter_t  dccp_sc_stats;
43 } dccp_stats_cpu_t;

45 /*
46  * MIB-II
47  */
48 #define DCCPS_BUMP_MIB(dccps, x) \
49     BUMP_MIB(&(dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_mib, x)

51 #define DCCPS_UPDATE_MIB(dccps, x, y) \
52     UPDATE_MIB(&(dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_mib, x, y)

54 #define DCCP_STAT(dccps, x) \
55     ((dccps)->dccps_sc[CPU->cpu_seqid]->dccp_sc_stats.x++)

57 #ifdef __cplusplus
58 }
59 #endif

61 #endif /* _INET_DCCP_DCCP_STATS_H */
```

new/usr/src/uts/common/inet/dccp_stats.h

2

62 #endif /* ! codereview */

new/usr/src/uts/common/inet/ip.h

1

```
*****
140318 Sat Aug 18 10:37:12 2012
new/usr/src/uts/common/inet/ip.h
dccp: starting module template
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 1990 Mentat Inc.
25  */

27 #ifndef _INET_IP_H
28 #define _INET_IP_H

30 #ifdef __cplusplus
31 extern "C" {
32 #endif

34 #include <sys/isa_defs.h>
35 #include <sys/types.h>
36 #include <inet/mib2.h>
37 #include <inet/nd.h>
38 #include <sys/atomic.h>
39 #include <net/if_dl.h>
40 #include <net/if.h>
41 #include <netinet/ip.h>
42 #include <netinet/igmp.h>
43 #include <sys/neti.h>
44 #include <sys/hook.h>
45 #include <sys/hook_event.h>
46 #include <sys/hook_impl.h>
47 #include <inet/ip_stack.h>

49 #ifdef _KERNEL
50 #include <netinet/ip6.h>
51 #include <sys/avl.h>
52 #include <sys/list.h>
53 #include <sys/vmem.h>
54 #include <sys/squeue.h>
55 #include <net/route.h>
56 #include <sys/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        IP_ABITS
100 #define IPV6_ABITS        128
101 #define IP_MAX_HW_LEN     40

103 #define IP_HOST_MASK      (ipaddr_t)0xffffffffu

105 #define IP_CSUM(mp, off, sum)      (~ip_cksum(mp, off, sum) & 0xFFFF)
106 #define IP_CSUM_PARTIAL(mp, off, sum)  ip_cksum(mp, off, sum)
107 #define IP_BCSUM_PARTIAL(bp, len, sum) bcksum(bp, len, sum)

109 #define ILL_FRAG_HASH_TBL_COUNT ((unsigned int)64)
110 #define ILL_FRAG_HASH_TBL_SIZE  (ILL_FRAG_HASH_TBL_COUNT * sizeof (ipfb_t))

112 #define IPV4_ADDR_LEN      4
113 #define IP_ADDR_LEN        IPV4_ADDR_LEN
114 #define IP_ARP_PROTO_TYPE  0x0800

116 #define IPV4_VERSION        4
117 #define IP_VERSION          IPV4_VERSION
118 #define IP_SIMPLE_HDR_LENGTH_IN_WORDS  5
119 #define IP_SIMPLE_HDR_LENGTH  20
120 #define IP_MAX_HDR_LENGTH    60

122 #define IP_MAX_OPT_LENGTH (IP_MAX_HDR_LENGTH-IP_SIMPLE_HDR_LENGTH)

124 #define IP_MIN_MTU          (IP_MAX_HDR_LENGTH + 8) /* 68 bytes */

126 /*
127  * XXX IP_MAXPACKET is defined in <netinet/ip.h> as well. At some point the
```

```

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 ipopty_s
168 {
169     uint8_t *ipopty_next; /* next option to look at */
170     uint8_t *ipopty_end; /* end of options */
171     uint8_t *ipopty_cur; /* start of current option */
172     uint8_t ipoptp_len; /* length of current option */
173     uint32_t ipoptp_flags;
174 } ipopty_t;

176 /*
177 * Flag(s) for ipopty_flags
178 */
179 #define IPOPTY_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 iocthref 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 recvd */
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_ILL         0x10   /* conn_update_ill 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 fmode 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_refhold */
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 addrs for pending req */
667     rtx_state_t      ilm_rtx;       /* SCR retransmission state */
668     ipaddr_t         ilm_ifaddr;    /* For IPv4 netstat */
669     ip_stack_t       *ilm_ipst;     /* Does not have a netstack_hold */
670 } ilm_t;

672 #define ilm_addr      V4_PART_OF_V6(ilm_v6addr)

674 /*
675  * Soft reference to an IPsec SA.
676  *
677  * On relative terms, conn's can be persistent (living as long as the
678  * processes which create them), while SA's are ephemeral (dying when
679  * they hit their time-based or byte-based lifetimes).
680  *
681  * We could hold a hard reference to an SA from an ipsec_latch_t,
682  * but this would cause expired SA's to linger for a potentially
683  * unbounded time.
684  *
685  * Instead, we remember the hash bucket number and bucket generation
686  * in addition to the pointer. The bucket generation is incremented on
687  * each deletion.
688  */
689 typedef struct ipsa_ref_s
690 {
691     struct ipsa_s     *ipsr_sa;
692     struct isaf_s     *ipsr_bucket;
693     uint64_t          ipsr_gen;
694 } ipsa_ref_t;

696 /*
697  * IPsec "latching" state.
698  *
699  * In the presence of IPsec policy, fully-bound conn's bind a connection
700  * to more than just the 5-tuple, but also a specific IPsec action and
701  * identity-pair.
702  * The identity pair is accessed from both the receive and transmit side
703  * hence it is maintained in the ipsec_latch_t structure. conn_latch and
704  * ixa_ipsec_latch points to it.
705  * The policy and actions are stored in conn_latch_in_policy and
706  * conn_latch_in_action for the inbound side, and in ixa_ipsec_policy and
707  * ixa_ipsec_action for the transmit side.
708  *
709  * As an optimization, we also cache soft references to IPsec SA's in
710  * ip_xmit_attr_t so that we can fast-path around most of the work needed for
711  * outbound IPsec SA selection.
712  */
713 typedef struct ipsec_latch_s
714 {
715     kmutex_t          ipl_lock;
716     uint32_t          ipl_refcnt;

718     struct ipsid_s     *ipl_local_cid;
719     struct ipsid_s     *ipl_remote_cid;
720     unsigned int       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_spipe;      /* 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_wscales_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 refholds 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;    /* Back pointer to our ill */
938     int ipif_id;              /* Logical unit number */
939     in6_addr_t ipif_v6lcl_addr; /* Local IP address for this if. */
940     in6_addr_t ipif_v6subnet;  /* Subnet prefix for this if. */
941     in6_addr_t ipif_v6net_mask; /* Net mask for this interface. */
942     in6_addr_t ipif_v6brd_addr; /* Broadcast addr for this interface. */
943     in6_addr_t ipif_v6pp_dst_addr; /* Point-to-point dest address. */
944     uint64_t ipif_flags;       /* Interface flags. */
945     uint_t ipif_ire_type;      /* IRE_LOCAL or IRE_LOOPBACK */

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; /* Inbound packets for our dead IREs */

953     /* Exclusive bit fields, protected by ipsq_t */
954     unsigned int
955         ipif_was_up : 1,    /* ipif was up before */
956         ipif_addr_ready : 1, /* DAD is done */
957         ipif_was_dup : 1,    /* DAD had failed */
958         ipif_added_nce : 1,  /* nce added for local address */

960         ipif_pad_to_31 : 28;

962     ilm_t *ipif_allhosts_ilm; /* For all-nodes join */
963     ilm_t *ipif_solmulti_ilm; /* For IPv6 solicited multicast join */

965     uint_t ipif_seqid;        /* unique index across all ill_s */
966     uint_t ipif_state_flags;  /* See IPIF_* flag defs above */
967     uint_t ipif_refcnt;       /* active consistent reader cnt */

969     zoneid_t ipif_zoneid;     /* zone ID number */
970     timeout_id_t ipif_recovery_id; /* Timer for DAD recovery */
971     boolean_t ipif_trace_disable; /* True when alloc fails */
972     /*
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 ill_s 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; /* bound ipif chain */
981     boolean_t ipif_bound; /* B_TRUE if we successfully bound */

983     struct ire_s *ipif_ire_local; /* Our IRE_LOCAL or LOOPBACK */
984     struct ire_s *ipif_ire_if;    /* Our IRE_INTERFACE */
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  * ipif_v6brd_addr
1015  * ipif_v6pp_dst_addr
1016  * ipif_flags                ill_lock              ill_lock
1017  * ipif_ire_type             ipsq + down ill        up ill
1018  *
1019  * ipif_ib_pkt_count         Approx
1020  *
1021  * bit fields                ill_lock              ill_lock
1022  *
1023  * ipif_allhosts_ilm         ipsq                  ipsq
1024  * ipif_solmulti_ilm         ipsq                  ipsq
1025  *
1026  * ipif_seqid                ipsq                  Write once
1027  *
1028  * ipif_state_flags          ill_lock              ill_lock
1029  * ipif_refcnt               ill_lock              ill_lock
1030  * ipif_bound_ill            ipsq + ipmp_lock      ipsq OR ipmp_lock
1031  * ipif_bound_next           ipsq                  ipsq
1032  * ipif_bound                 ipsq                  ipsq
1033  *
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 #ifndef 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 have
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_mphhead      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_mphhead            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_mphhead; /* 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_mphhead; /* 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(aligned_struct)      P2ROUNDUP(sizeof (struct aligned_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 IRB_MARK_CONDEMNED 0x0001 /* Contains some IRE_IS_CONDEMNED */
1276 #define IRB_MARK_DYNAMIC 0x0002 /* Dynamically allocated */
1277 /* Once IPv6 uses radix then IRB_MARK_DYNAMIC will be always be set */
1278     uint_t             irb_ire_cnt;              /* Num of active IRE in this bucket */
1279     int                irb_nire;                 /* Num of ftable ire's that ref irb */
1280     ip_stack_t         *irb_ipst;                /* Does not have a netstack_hold */
1281 } irb_t;

1283 /*
1284  * This is the structure used to store the multicast physical addresses
1285  * that an interface has joined.
1286  * The refcnt keeps track of the number of multicast IP addresses mapping
1287  * to a physical multicast address.
1288  */
1289 typedef struct multiphysaddr_s {
1290     struct multiphysaddr_s *mpa_next;
1291     char mpa_addr[IP_MAX_HW_LEN];
1292     int mpa_refcnt;
1293 } multiphysaddr_t;

1295 #define IRB2RT(irb)      (rt_t *)((caddr_t)(irb) - offsetof(rt_t, rt_irb))

1297 /* Forward declarations */
1298 struct dce_s;
1299 typedef struct dce_s dce_t;
1300 struct ire_s;
1301 typedef struct ire_s ire_t;
1302 struct ncec_s;
1303 typedef struct ncec_s ncec_t;
1304 struct nce_s;
1305 typedef struct nce_s nce_t;
1306 struct ip_recv_attr_s;
1307 typedef struct ip_recv_attr_s ip_recv_attr_t;
1308 struct ip_xmit_attr_s;
1309 typedef struct ip_xmit_attr_s ip_xmit_attr_t;

1311 struct tsol_ire_gw_secattr_s;
1312 typedef struct tsol_ire_gw_secattr_s tsol_ire_gw_secattr_t;

1314 /*
1315  * This is a structure for a one-element route cache that is passed

```

```

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]_slowtimo */
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_zeroCopy_capab_s ill_zeroCopy_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_CLI_ILL(ill) \
1467     (((ill)->ill_usersrc_ifindex != 0) && \
1468      ((ill)->ill_usersrc_grp_next != NULL))

1470 /* Is this an 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_actif               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_actif; /* 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 in V4 group */
1563     uint_t         gr_nv6; /* number of ill in V6 group */
1564     uint_t         gr_pendv4; /* number of pending ill in V4 group */
1565     uint_t         gr_pendv6; /* number of pending ill 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_proxyarp; /* 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_scalar_t ill_sap_length; /* Including sign (for position) */
1636     uint_t   ill_phys_addr_length; /* Excluding the sap. */
1637     uint_t   ill_bcast_addr_length; /* Only set when the DL provider */
1638     /* supports broadcast. */
1639     t_uscalar_t ill_mactype;
1640     uint8_t *ill_frag_ptr; /* Reassembly state. */
1641     timeout_id_t ill_frag_timer_id; /* timeout id for the frag timer */
1642     ipfb_t *ill_frag_hash_tbl; /* Fragment hash list head. */

1644     krwlock_t ill_mcast_lock; /* Protects multicast state */
1645     kmutex_t ill_mcast_serializer; /* Serialize across ilg and ilm state */

```

```

1646     ilm_t      *ill_ilm; /* Multicast membership for ill */
1647     uint_t      ill_global_timer; /* for IGMPv3/MLDv2 general queries */
1648     int         ill_mcast_type; /* type of router which is querier */
1649     /* on this interface */
1650     uint16_t    ill_mcast_v1_time; /* # slow timeouts since last v1 qry */
1651     uint16_t    ill_mcast_v2_time; /* # slow timeouts since last v2 qry */
1652     uint8_t     ill_mcast_v1_tset; /* 1 => timer is set; 0 => not set */
1653     uint8_t     ill_mcast_v2_tset; /* 1 => timer is set; 0 => not set */

1655     uint8_t     ill_mcast_rv; /* IGMPv3/MLDv2 robustness variable */
1656     int         ill_mcast_qi; /* IGMPv3/MLDv2 query interval var */

1658     /*
1659      * All non-NULL cells between 'ill_first_mp_to_free' and
1660      * 'ill_last_mp_to_free' are freed in ill_delete.
1661      */
1662     #define ill_first_mp_to_free    ill_bcast_mp
1663     mblk_t      *ill_bcast_mp; /* DLPI header for broadcasts. */
1664     mblk_t      *ill_unbind_mp; /* unbind mp from ill_dl_up() */
1665     mblk_t      *ill_promiscoff_mp; /* for ill_leave_allmulti() */
1666     mblk_t      *ill_dlpi_deferred; /* b_next chain of control messages */
1667     mblk_t      *ill_dest_addr_mp; /* mblk which holds ill_dest_addr */
1668     mblk_t      *ill_replumb_mp; /* replumb mp from ill_replumb() */
1669     mblk_t      *ill_phys_addr_mp; /* mblk which holds ill_phys_addr */
1670     mblk_t      *ill_mcast_deferred; /* b_next chain of IGMP/MLD packets */
1671     #define ill_last_mp_to_free    ill_mcast_deferred

1673     cred_t      *ill_credp; /* opener's credentials */
1674     uint8_t      *ill_phys_addr; /* ill_phys_addr_mp->b_rptr + off */
1675     uint8_t      *ill_dest_addr; /* ill_dest_addr_mp->b_rptr + off */

1677     uint_t      ill_state_flags; /* see ILL_* flags above */

1679     /* Following bit fields protected by ipsq_t */
1680     uint_t
1681         ill_needs_attach : 1,
1682         ill_reserved : 1,
1683         ill_isv6 : 1,
1684         ill_dlpi_style_set : 1,

1686         ill_ifname_pending : 1,
1687         ill_logical_down : 1,
1688         ill_dl_up : 1,
1689         ill_up_ipifs : 1,

1691         ill_note_link : 1, /* supports link-up notification */
1692         ill_capab_reneg : 1, /* capability renegotiation to be done */
1693         ill_dld_capab_inprog : 1, /* direct dld capab call in prog */
1694         ill_need_recover_multicast : 1,

1696         ill_replumbing : 1,
1697         ill_arl_dlpi_pending : 1,
1698         ill_grp_pending : 1,

1700         ill_pad_to_bit_31 : 17;

1702     /* Following bit fields protected by ill_lock */
1703     uint_t
1704         ill_fragtimer_executing : 1,
1705         ill_fragtimer_needrestart : 1,
1706         ill_manual_token : 1, /* system won't override ill_token */
1707     /*
1708      * ill_manual_linklocal : system will not change the
1709      * linklocal whenever ill_token changes.
1710      */
1711     ill_manual_linklocal : 1,

```

```

1713         ill_manual_dst_linklocal : 1, /* same for pt-pt dst linklocal */
1715         ill_pad_bit_31 : 27;

1717 /*
1718  * Used in SIOCSIFMUXID and SIOCGIFMUXID for 'ifconfig unplumb'.
1719  */
1720 int         ill_muxid;          /* muxid returned from plink */

1722 /* Used for IP frag reassembly throttling on a per ILL basis. */
1723 uint_t     ill_ipf_gen;        /* Generation of next fragment queue */
1724 uint_t     ill_frag_count;     /* Count of all reassembly mblk bytes */
1725 uint_t     ill_frag_free_num_pkts; /* num of fragmented packets to free */
1726 clock_t    ill_last_frag_clean_time; /* time when frag's were pruned */
1727 int         ill_type;          /* From <net/if_types.h> */
1728 uint_t     ill_dlpi_multicast_state; /* See below IDS_* */
1729 uint_t     ill_dlpi_fastpath_state; /* See below IDS_* */

1731 /*
1732  * Capabilities related fields.
1733  */
1734 uint_t     ill_dlpi_capab_state; /* State of capability query, IDCS_* */
1735 uint_t     ill_capab_pending_cnt;
1736 uint64_t   ill_capabilities; /* Enabled capabilities, ILL_CAPAB_* */
1737 ill_hcksum_capab_t *ill_hcksum_capab; /* H/W cksunning 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_lls 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_lls_mp; /* mblk which holds ill_nd_lls */
1765 uint8_t       *ill_nd_lls; /* Link Layer Address */
1766 uint_t       ill_nd_lls_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
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)
1851
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_lla_mp        ipsq + down ill           only when ill is up
1937 * ill_nd_lla           ipsq + down ill           only when ill is up
1938 * ill_nd_lla_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              ill_lock                 ill_lock
1950 * ill_src_ipif         ill_g_lock                 ill_g_lock
1951 * ill_trace            ill_lock                 ill_lock
1952 * ill_usersrc_grp_next ill_g_usersrc_lock        ill_g_usersrc_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_grpnod           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_mroutercnt        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 ixaf_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 ixaf_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     queue_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     ixaf_notify_t   *ixa_notify;         /* Registered upcall notify function */
2204     void            *ixa_notify_cookie;  /* ULP cookie for ixaf_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 /* ixaf_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 /* ixaf_pmtu/ixaf_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 /* ixaf_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 /* ixaf_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    (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_sq;
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_sq;
2321     mblk_t         *ira_target_sq_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 chksum */

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 readown */
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     krmlock_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     pfirmecv_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_OOBINLINE_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]ifreq ioctl's */
2923     sin_t     *ci_sin;        /* the sin struct passed down */
2924     sin6_t    *ci_sin6;       /* the sin6_t struct passed down */
2925     struct lifreq *ci_lifr;    /* the lifreq struct passed down */
2926 } cmd_info_t;

2928 extern struct kmem_cache *ire_cache;

2930 extern ipaddr_t ip_g_all_ones;

2932 extern uint_t    ip_loopback_mtu;        /* /etc/system */
2933 extern uint_t    ip_loopback_mtuplus;
2934 extern uint_t    ip_loopback_mtu_v6plus;

2936 extern vmem_t    *ip_minor_arena_sa;
2937 extern vmem_t    *ip_minor_arena_la;

2939 /*
2940  * ip_g_forward controls IP forwarding. It takes two values:
2941  * 0: IP_FORWARD_NEVER    Don't forward packets ever.
2942  * 1: IP_FORWARD_ALWAYS   Forward packets for elsewhere.
2943  *
2944  * RFC1122 says there must be a configuration switch to control forwarding,
2945  * but that the default MUST be to not forward packets ever. Implicit
2946  * control based on configuration of multiple interfaces MUST NOT be
2947  * implemented (Section 3.1). SunOS 4.1 did provide the "automatic" capability
2948  * and, in fact, it was the default. That capability is now provided in the
2949  * /etc/rc2.d/S69inet script.
2950  */

2952 #define ips_ip_respond_to_address_mask_broadcast \
2953     ips_propinfo_tbl[0].prop_cur_bval
2954 #define ips_ip_g_resp_to_echo_bcast    ips_propinfo_tbl[1].prop_cur_bval
2955 #define ips_ip_g_resp_to_echo_mcast    ips_propinfo_tbl[2].prop_cur_bval
2956 #define ips_ip_g_resp_to_timestamp     ips_propinfo_tbl[3].prop_cur_bval
2957 #define ips_ip_g_resp_to_timestamp_bcast ips_propinfo_tbl[4].prop_cur_bval
2958 #define ips_ip_g_send_redirects        ips_propinfo_tbl[5].prop_cur_bval
2959 #define ips_ip_g_forward_directed_bcast ips_propinfo_tbl[6].prop_cur_bval
2960 #define ips_ip_mrtdebug                ips_propinfo_tbl[7].prop_cur_uval
2961 #define ips_ip_ire_reclaim_fraction    ips_propinfo_tbl[8].prop_cur_uval
2962 #define ips_ip_nce_reclaim_fraction    ips_propinfo_tbl[9].prop_cur_uval
2963 #define ips_ip_dce_reclaim_fraction    ips_propinfo_tbl[10].prop_cur_uval
2964 #define ips_ip_def_ttl                 ips_propinfo_tbl[11].prop_cur_uval
2965 #define ips_ip_forward_src_routed      ips_propinfo_tbl[12].prop_cur_bval

```

```

2966 #define ips_ip_wroff_extra             ips_propinfo_tbl[13].prop_cur_uval
2967 #define ips_ip_pathmtu_interval        ips_propinfo_tbl[14].prop_cur_uval
2968 #define ips_ip_icmp_return             ips_propinfo_tbl[15].prop_cur_uval
2969 #define ips_ip_path_mtu_discovery      ips_propinfo_tbl[16].prop_cur_bval
2970 #define ips_ip_pmtu_min                ips_propinfo_tbl[17].prop_cur_uval
2971 #define ips_ip_ignore_redirect         ips_propinfo_tbl[18].prop_cur_bval
2972 #define ips_ip_arp_icmp_error          ips_propinfo_tbl[19].prop_cur_bval
2973 #define ips_ip_broadcast_ttl           ips_propinfo_tbl[20].prop_cur_uval
2974 #define ips_ip_icmp_err_interval       ips_propinfo_tbl[21].prop_cur_uval
2975 #define ips_ip_icmp_err_burst          ips_propinfo_tbl[22].prop_cur_uval
2976 #define ips_ip_reass_queue_bytes       ips_propinfo_tbl[23].prop_cur_uval
2977 #define ips_ip_strict_dst_multihoming  ips_propinfo_tbl[24].prop_cur_uval
2978 #define ips_ip_adrrs_per_if            ips_propinfo_tbl[25].prop_cur_uval
2979 #define ips_ipsec_override_persocket_policy ips_propinfo_tbl[26].prop_cur_bval
2980 #define ips_icmp_accept_clear_messages ips_propinfo_tbl[27].prop_cur_bval
2981 #define ips_igmp_accept_clear_messages ips_propinfo_tbl[28].prop_cur_bval

2983 /* IPv6 configuration knobs */
2984 #define ips_delay_first_probe_time     ips_propinfo_tbl[29].prop_cur_uval
2985 #define ips_max_unicast_solicit        ips_propinfo_tbl[30].prop_cur_uval
2986 #define ips_ipv6_def_hops              ips_propinfo_tbl[31].prop_cur_uval
2987 #define ips_ipv6_icmp_return           ips_propinfo_tbl[32].prop_cur_uval
2988 #define ips_ipv6_forward_src_routed    ips_propinfo_tbl[33].prop_cur_bval
2989 #define ips_ipv6_resp_echo_mcast      ips_propinfo_tbl[34].prop_cur_bval
2990 #define ips_ipv6_send_redirects        ips_propinfo_tbl[35].prop_cur_bval
2991 #define ips_ipv6_ignore_redirect       ips_propinfo_tbl[36].prop_cur_bval
2992 #define ips_ipv6_strict_dst_multihoming ips_propinfo_tbl[37].prop_cur_uval
2993 #define ips_src_check                  ips_propinfo_tbl[38].prop_cur_uval
2994 #define ips_ipsec_policy_log_interval  ips_propinfo_tbl[39].prop_cur_uval
2995 #define ips_pim_accept_clear_messages  ips_propinfo_tbl[40].prop_cur_bval
2996 #define ips_ip_ndp_unsolicit_interval  ips_propinfo_tbl[41].prop_cur_uval
2997 #define ips_ip_ndp_unsolicit_count     ips_propinfo_tbl[42].prop_cur_uval
2998 #define ips_ipv6_ignore_home_address_opt ips_propinfo_tbl[43].prop_cur_bval

3000 /* Misc IP configuration knobs */
3001 #define ips_ip_policy_mask              ips_propinfo_tbl[44].prop_cur_uval
3002 #define ips_ip_ecmp_behavior            ips_propinfo_tbl[45].prop_cur_uval
3003 #define ips_ip_multirt_ttl              ips_propinfo_tbl[46].prop_cur_uval
3004 #define ips_ip_ire_badcnt_lifetime      ips_propinfo_tbl[47].prop_cur_uval
3005 #define ips_ip_max_temp_idle            ips_propinfo_tbl[48].prop_cur_uval
3006 #define ips_ip_max_temp_defend          ips_propinfo_tbl[49].prop_cur_uval
3007 #define ips_ip_max_defend               ips_propinfo_tbl[50].prop_cur_uval
3008 #define ips_ip_defend_interval          ips_propinfo_tbl[51].prop_cur_uval
3009 #define ips_ip_dup_recovery             ips_propinfo_tbl[52].prop_cur_uval
3010 #define ips_ip_restrict_interzone_loopback ips_propinfo_tbl[53].prop_cur_bval
3011 #define ips_ip_lso_outbound             ips_propinfo_tbl[54].prop_cur_bval
3012 #define ips_igmp_max_version            ips_propinfo_tbl[55].prop_cur_uval
3013 #define ips_mld_max_version             ips_propinfo_tbl[56].prop_cur_uval
3014 #define ips_ip_forwarding               ips_propinfo_tbl[57].prop_cur_bval
3015 #define ips_ipv6_forwarding             ips_propinfo_tbl[58].prop_cur_bval
3016 #define ips_ip_reassembly_timeout       ips_propinfo_tbl[59].prop_cur_uval
3017 #define ips_ipv6_reassembly_timeout     ips_propinfo_tbl[60].prop_cur_uval
3018 #define ips_ip_cgtp_filter              ips_propinfo_tbl[61].prop_cur_bval
3019 #define ips_arp_probe_delay             ips_propinfo_tbl[62].prop_cur_uval
3020 #define ips_arp_fastprobe_delay         ips_propinfo_tbl[63].prop_cur_uval
3021 #define ips_arp_probe_interval          ips_propinfo_tbl[64].prop_cur_uval
3022 #define ips_arp_fastprobe_interval      ips_propinfo_tbl[65].prop_cur_uval
3023 #define ips_arp_probe_count             ips_propinfo_tbl[66].prop_cur_uval
3024 #define ips_arp_fastprobe_count          ips_propinfo_tbl[67].prop_cur_uval
3025 #define ips_ipv4_dad_announce_interval  ips_propinfo_tbl[68].prop_cur_uval
3026 #define ips_ipv6_dad_announce_interval ips_propinfo_tbl[69].prop_cur_uval
3027 #define ips_arp_defend_interval         ips_propinfo_tbl[70].prop_cur_uval
3028 #define ips_arp_defend_rate             ips_propinfo_tbl[71].prop_cur_uval
3029 #define ips_ndp_defend_interval         ips_propinfo_tbl[72].prop_cur_uval
3030 #define ips_ndp_defend_rate             ips_propinfo_tbl[73].prop_cur_uval
3031 #define ips_arp_defend_period            ips_propinfo_tbl[74].prop_cur_uval

```

```

3032 #define ips_ndp_defend_period      ips_propinfo_tbl[75].prop_cur_uval
3033 #define ips_ipv4_icmp_return_pmtu  ips_propinfo_tbl[76].prop_cur_bval
3034 #define ips_ipv6_icmp_return_pmtu  ips_propinfo_tbl[77].prop_cur_bval
3035 #define ips_ip_arp_publish_count    ips_propinfo_tbl[78].prop_cur_uval
3036 #define ips_ip_arp_publish_interval ips_propinfo_tbl[79].prop_cur_uval
3037 #define ips_ip_strict_src_multihoming ips_propinfo_tbl[80].prop_cur_uval
3038 #define ips_ipv6_strict_src_multihoming ips_propinfo_tbl[81].prop_cur_uval
3039 #define ips_ipv6_drop_inbound_icmpv6 ips_propinfo_tbl[82].prop_cur_bval

3041 extern int      dohwcksum;        /* use h/w cksum if supported by the h/w */
3042 #ifdef ZC_TEST
3043 extern int      noswcksum;
3044 #endif

3046 extern char      ipif_loopback_name[];

3048 extern nv_t      *ire_nv_tbl;

3050 extern struct module_info ip_mod_info;

3052 #define HOOKS4_INTERESTED_PHYSICAL_IN(ipst) \
3053 ((ipst)->ips_ip4_physical_in_event.he_interested)
3054 #define HOOKS6_INTERESTED_PHYSICAL_IN(ipst) \
3055 ((ipst)->ips_ip6_physical_in_event.he_interested)
3056 #define HOOKS4_INTERESTED_PHYSICAL_OUT(ipst) \
3057 ((ipst)->ips_ip4_physical_out_event.he_interested)
3058 #define HOOKS6_INTERESTED_PHYSICAL_OUT(ipst) \
3059 ((ipst)->ips_ip6_physical_out_event.he_interested)
3060 #define HOOKS4_INTERESTED_FORWARDING(ipst) \
3061 ((ipst)->ips_ip4_forwarding_event.he_interested)
3062 #define HOOKS6_INTERESTED_FORWARDING(ipst) \
3063 ((ipst)->ips_ip6_forwarding_event.he_interested)
3064 #define HOOKS4_INTERESTED_LOOPBACK_IN(ipst) \
3065 ((ipst)->ips_ip4_loopback_in_event.he_interested)
3066 #define HOOKS6_INTERESTED_LOOPBACK_IN(ipst) \
3067 ((ipst)->ips_ip6_loopback_in_event.he_interested)
3068 #define HOOKS4_INTERESTED_LOOPBACK_OUT(ipst) \
3069 ((ipst)->ips_ip4_loopback_out_event.he_interested)
3070 #define HOOKS6_INTERESTED_LOOPBACK_OUT(ipst) \
3071 ((ipst)->ips_ip6_loopback_out_event.he_interested)
3072 /*
3073  * Hooks marcos used inside of ip
3074  * The callers use the above INTERESTED macros first, hence
3075  * the he_interested check is 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 *, squeue_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 *, squeue_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 *, squeue_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_multihome(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_massage_options(ipha_t *, netstack_t *);
3370 extern ipaddr_t  ip_net_mask(ipaddr_t);
3371 extern void      arp_bringsup_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 #ifdef DEBUG
3441 extern boolean_t th_trace_ref(const void *, ip_stack_t *);
3442 extern void     th_trace_unref(const void *);
3443 extern void     th_trace_cleanup(const void *, boolean_t);
3444 extern void     ire_trace_ref(ire_t *);
3445 extern void     ire_untrace_ref(ire_t *);
3446 #endif

3448 extern int     ip_srcid_insert(const in6_addr_t *, zoneid_t, ip_stack_t *);
3449 extern int     ip_srcid_remove(const in6_addr_t *, zoneid_t, ip_stack_t *);
3450 extern void     ip_srcid_find_id(uint_t, in6_addr_t *, zoneid_t, netstack_t *);
3451 extern uint_t   ip_srcid_find_addr(const in6_addr_t *, zoneid_t, netstack_t *);

3453 extern uint8_t ipoptp_next(ipoptp_t *);
3454 extern uint8_t ipoptp_first(ipoptp_t *, ipha_t *);
3455 extern int     ip_opt_get_user(conn_t *, uchar_t *);
3456 extern int     ipsec_req_from_conn(conn_t *, ipsec_req_t *, int);
3457 extern int     ip_snmp_get(queue_t *q, mblk_t *mctl, int level, boolean_t);
3458 extern int     ip_snmp_set(queue_t *q, int, int, uchar_t *, int);
3459 extern void     ip_process_ioctl(ipsq_t *, queue_t *, mblk_t *, void *);
3460 extern void     ip_quiesce_conn(conn_t *);
3461 extern void     ip_reprocess_ioctl(ipsq_t *, queue_t *, mblk_t *, void *);
3462 extern void     ip_ioctl_finish(queue_t *, mblk_t *, int, int, ipsq_t *);

3464 extern boolean_t ip_cmpbuf(const void *, uint_t, boolean_t, const void *,
3465     uint_t);
3466 extern boolean_t ip_allocbuf(void **, uint_t *, boolean_t, const void *,
3467     uint_t);
3468 extern void     ip_savebuf(void **, uint_t *, boolean_t, const void *, uint_t);

3470 extern boolean_t ipsq_pending_mp_cleanup(ill_t *, conn_t *);
3471 extern void     conn_ioctl_cleanup(conn_t *);

3473 extern void     ip_unbind(conn_t *);

3475 extern void     tnet_init(void);
3476 extern void     tnet_fini(void);

3478 /*
3479  * Hook functions to enable cluster networking
3480  * On non-clustered systems these vectors must always be NULL.
3481  */
3482 extern int (*cl_inet_isclusterwide)(netstackid_t stack_id, uint8_t protocol,
3483     sa_family_t addr_family, uint8_t *laddrp, void *args);
3484 extern uint32_t (*cl_inet_ipident)(netstackid_t stack_id, uint8_t protocol,
3485     sa_family_t addr_family, uint8_t *laddrp, uint8_t *faddrp,
3486     void *args);
3487 extern int (*cl_inet_connect2)(netstackid_t stack_id, uint8_t protocol,
3488     boolean_t is_outgoing, sa_family_t addr_family, uint8_t *laddrp,
3489     in_port_t lport, uint8_t *faddrp, in_port_t fport, void *args);
3490 extern void (*cl_inet_getspi)(netstackid_t, uint8_t, uint8_t *, size_t,
3491     void *);
3492 extern void (*cl_inet_getspi)(netstackid_t stack_id, uint8_t protocol,
3493     uint8_t *ptr, size_t len, void *args);

```

```

3494 extern int (*cl_inet_checkspi)(netstackid_t stack_id, uint8_t protocol,
3495     uint32_t spi, void *args);
3496 extern void (*cl_inet_deletespi)(netstackid_t stack_id, uint8_t protocol,
3497     uint32_t spi, void *args);
3498 extern void (*cl_inet_idlesa)(netstackid_t, uint8_t, uint32_t,
3499     sa_family_t, in6_addr_t, in6_addr_t, void *);

3502 /* Hooks for CGTP (multirt routes) filtering module */
3503 #define CGTP_FILTER_REV_1      1
3504 #define CGTP_FILTER_REV_2      2
3505 #define CGTP_FILTER_REV_3      3
3506 #define CGTP_FILTER_REV        CGTP_FILTER_REV_3

3508 /* cfo_filter and cfo_filter_v6 hooks return values */
3509 #define CGTP_IP_PKT_NOT_CGTP    0
3510 #define CGTP_IP_PKT_PREMIUM     1
3511 #define CGTP_IP_PKT_DUPLICATE   2

3513 /* Version 3 of the filter interface */
3514 typedef struct cgtp_filter_ops {
3515     int     cfo_filter_rev; /* CGTP_FILTER_REV_3 */
3516     int     (*cfo_change_state)(netstackid_t, int);
3517     int     (*cfo_add_dest_v4)(netstackid_t, ipaddr_t, ipaddr_t,
3518         ipaddr_t, ipaddr_t);
3519     int     (*cfo_del_dest_v4)(netstackid_t, ipaddr_t, ipaddr_t);
3520     int     (*cfo_add_dest_v6)(netstackid_t, in6_addr_t *, in6_addr_t *,
3521         in6_addr_t *, in6_addr_t *);
3522     int     (*cfo_del_dest_v6)(netstackid_t, in6_addr_t *, in6_addr_t *);
3523     int     (*cfo_filter)(netstackid_t, uint_t, mblk_t *);
3524     int     (*cfo_filter_v6)(netstackid_t, uint_t, ip6_t *,
3525         ip6_frag_t *);
3526 } cgtp_filter_ops_t;

3528 #define CGTP_MCAST_SUCCESS      1

3530 /*
3531  * The separate CGTP module needs this global symbol so that it
3532  * can check the version and determine whether to use the old or the new
3533  * version of the filtering interface.
3534  */
3535 extern int     ip_cgtp_filter_rev;

3537 extern int     ip_cgtp_filter_supported(void);
3538 extern int     ip_cgtp_filter_register(netstackid_t, cgtp_filter_ops_t *);
3539 extern int     ip_cgtp_filter_unregister(netstackid_t);
3540 extern int     ip_cgtp_filter_is_registered(netstackid_t);

3542 /*
3543  * rr_ring_state cycles in the order shown below from RR_FREE through
3544  * RR_FREE_IN_PROG and back to RR_FREE.
3545  */
3546 typedef enum {
3547     RR_FREE, /* Free slot */
3548     RR_SQUEUE_UNBOUND, /* Ring's squeue is unbound */
3549     RR_SQUEUE_BIND_INPROG, /* Ring's squeue bind in progress */
3550     RR_SQUEUE_BOUND, /* Ring's squeue bound to cpu */
3551     RR_FREE_INPROG /* Ring is being freed */
3552 } ip_ring_state_t;

3554 #define ILL_MAX_RINGS          256 /* Max num of rx rings we can manage */
3555 #define ILL_POLLING            0x01 /* Polling in use */

3557 /*
3558  * These functions pointer types are exported by the mac/dls layer.
3559  * we need to duplicate the definitions here because we cannot

```

```

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);
3572
3573 /*
3574  * POLLING README
3575  * sq_get_pkts() is called to pick packets from softring 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  */
3580
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     queue_t *, mblk_t *, mblk_t **, uint_t *);
3588
3589 /*
3590  * rr_intr_enable, rr_intr_disable, rr_rx_handle, rr_rx:
3591  * May be accessed while in the queue 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     queue_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 };
3607
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;
3621
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 };
3634
3635 /*
3636  * IP squeues exports
3637  */
3638 extern boolean_t ip_squeue_fanout;
3639
3640 #define IP_SQUEUE_GET(hint) ip_squeue_random(hint)
3641
3642 extern void ip_squeue_init(void (*)(queue_t *));
3643 extern queue_t *ip_squeue_random(uint_t);
3644 extern queue_t *ip_squeue_get(ill_rx_ring_t *);
3645 extern queue_t *ip_squeue_getfree(priority_t);
3646 extern int ip_squeue_cpu_move(queue_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 *);
3654
3655 extern void tcp_wput(queue_t *, mblk_t *);
3656
3657 extern int ip_fill_mtuinfo(conn_t *, ip_xmit_attr_t *,
3658     struct ip6_mtuinfo *);
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 *);
3664
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 dce_t *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);
3690
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_OUTPUT 47
3760 #define SQTAG_DCCP_CONN_REQ_UNBOUND 48
3761 #define SQTAG_DCCP_SEND_RESPONSE 49
3762 #define SQTAG_IP_DCCP_CLOSE 50
3763 #define SQTAG_DCCP_TIMER 51
3764 #define SQTAG_DCCP_SHUTDOWN_OUTPUT 52
3765 #endif /* ! codereview */

3767 extern sin_t     sin_null; /* Zero address for quick clears */
3768 extern sin6_t    sin6_null; /* Zero address for quick clears */

3770 #endif /* _KERNEL */

3772 #ifdef __cplusplus
3773 }
3774 #endif

3776 #endif /* _INET_IP_H */

```

new/usr/src/uts/common/inet/ip/ip.c

1

449431 Sat Aug 18 10:37:13 2012

new/usr/src/uts/common/inet/ip/ip.c

dccp: starting module template

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
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/pattr.h>
114 #include <inet/dccp.h>
115 #include <inet/dccp_impl.h>
116 #include <inet/dccp_ip.h>
117 #endif /* ! codereview */
118 #include <inet/ipclassifier.h>
119 #include <inet/sctp_ip.h>
120 #include <inet/sctp/sctp_impl.h>
121 #include <inet/udp_impl.h>
122 #include <inet/rawip_impl.h>
123 #include <inet/rts_impl.h>
124
125 #include <sys/tsol/label.h>
126 #include <sys/tsol/tnet.h>
```

new/usr/src/uts/common/inet/ip/ip.c

3

```
128 #include <sys/squeue_impl.h>
129 #include <inet/ip_arp.h>

131 #include <sys/clock_impl.h>      /* For LBOLT_FASTPATH{,64} */

133 /*
134  * Values for squeue switch:
135  * IP_SQUEUE_ENTER_NODRAIN: SQ_NODRAIN
136  * IP_SQUEUE_ENTER: SQ_PROCESS
137  * IP_SQUEUE_FILL: SQ_FILL
138  */
139 int ip_squeue_enter = IP_SQUEUE_ENTER; /* Setable in /etc/system */

141 int ip_squeue_flag;

143 /*
144  * Setable in /etc/system
145  */
146 int ip_poll_normal_ms = 100;
147 int ip_poll_normal_ticks = 0;
148 int ip_modclose_ackwait_ms = 3000;

150 /*
151  * It would be nice to have these present only in DEBUG systems, but the
152  * current design of the global symbol checking logic requires them to be
153  * unconditionally present.
154  */
155 uint_t ip_thread_data;          /* TSD key for debug support */
156 krwlock_t ip_thread_rwlock;
157 list_t ip_thread_list;

159 /*
160  * Structure to represent a linked list of msgblks. Used by ip_snmp_ functions.
161  */

163 struct listptr_s {
164     mblk_t *lp_head;      /* pointer to the head of the list */
165     mblk_t *lp_tail;      /* pointer to the tail of the list */
166 };

168 typedef struct listptr_s listptr_t;

170 /*
171  * This is used by ip_snmp_get_mib2_ip_route_media and
172  * ip_snmp_get_mib2_ip6_route_media to carry the lists of return data.
173  */
174 typedef struct iproutedata_s {
175     uint_t ird_idx;
176     uint_t ird_flags;      /* see below */
177     listptr_t ird_route;   /* ipRouteEntryTable */
178     listptr_t ird_netmedia; /* ipNetToMediaEntryTable */
179     listptr_t ird_attr;    /* ipRouteAttributeTable */
180 } iproutedata_t;

182 /* Include ire_testhidden and IRE_IF_CLONE routes */
183 #define IRD_REPORT_ALL 0x01

185 /*
186  * Cluster specific hooks. These should be NULL when booted as a non-cluster
187  */

189 /*
190  * Hook functions to enable cluster networking
191  * On non-clustered systems these vectors must always be NULL.
192  */
193 /* Hook function to Check ip specified ip address is a shared ip address
```

new/usr/src/uts/common/inet/ip/ip.c

4

```
194 * in the cluster
195 *
196 */
197 int (*cl_inet_isclusterwide)(netstackid_t stack_id, uint8_t protocol,
198     sa_family_t addr_family, uint8_t *laddrp, void *args) = NULL;

200 /*
201  * Hook function to generate cluster wide ip fragment identifier
202  */
203 uint32_t (*cl_inet_ipident)(netstackid_t stack_id, uint8_t protocol,
204     sa_family_t addr_family, uint8_t *laddrp, uint8_t *faddrp,
205     void *args) = NULL;

207 /*
208  * Hook function to generate cluster wide SPI.
209  */
210 void (*cl_inet_getspi)(netstackid_t, uint8_t, uint8_t *, size_t,
211     void *) = NULL;

213 /*
214  * Hook function to verify if the SPI is already utilized.
215  */

217 int (*cl_inet_checkspi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

219 /*
220  * Hook function to delete the SPI from the cluster wide repository.
221  */

223 void (*cl_inet_deletespi)(netstackid_t, uint8_t, uint32_t, void *) = NULL;

225 /*
226  * Hook function to inform the cluster when packet received on an IDLE SA
227  */

229 void (*cl_inet_idlesa)(netstackid_t, uint8_t, uint32_t, sa_family_t,
230     in6_addr_t, in6_addr_t, void *) = NULL;

232 /*
233  * Synchronization notes:
234  *
235  * IP is a fully D_MP STREAMS module/driver. Thus it does not depend on any
236  * MT level protection given by STREAMS. IP uses a combination of its own
237  * internal serialization mechanism and standard Solaris locking techniques.
238  * The internal serialization is per phyint. This is used to serialize
239  * plumbing operations, IPMP operations, most set ioctls, etc.
240  *
241  * Plumbing is a long sequence of operations involving message
242  * exchanges between IP, ARP and device drivers. Many set ioctls are typically
243  * involved in plumbing operations. A natural model is to serialize these
244  * ioctls one per ill. For example plumbing of hme0 and qfe0 can go on in
245  * parallel without any interference. But various set ioctls on hme0 are best
246  * serialized, along with IPMP operations and processing of DLPI control
247  * messages received from drivers on a per phyint basis. This serialization is
248  * provided by the ipsq_t and primitives operating on this. Details can
249  * be found in ip_if.c above the core primitives operating on ipsq_t.
250  *
251  * Lookups of an ipif or ill by a thread return a refheld ipif / ill.
252  * Similarly lookup of an ire by a thread also returns a refheld ire.
253  * In addition ipif's and ill's referenced by the ire are also indirectly
254  * refheld. Thus no ipif or ill can vanish as long as an ipif is refheld
255  * directly or indirectly. For example an SIOCIFADDR ioctl that changes the
256  * address of an ipif has to go through the ipsq_t. This ensures that only
257  * one such exclusive operation proceeds at any time on the ipif. It then
258  * waits for all refcnts
259  * associated with this ipif to come down to zero. The address is changed
```

```

260 * only after the ipif has been quiesced. Then the ipif is brought up again.
261 * More details are described above the comment in ip_sioctl_flags.
262 *
263 * Packet processing is based mostly on IREs and are fully multi-threaded
264 * using standard Solaris MT techniques.
265 *
266 * There are explicit locks in IP to handle:
267 * - The ip_g_head list maintained by mi_open_link() and friends.
268 *
269 * - The reassembly data structures (one lock per hash bucket)
270 *
271 * - conn_lock is meant to protect conn_t fields. The fields actually
272 *   protected by conn_lock are documented in the conn_t definition.
273 *
274 * - ire_lock to protect some of the fields of the ire, IRE tables
275 *   (one lock per hash bucket). Refer to ip_ire.c for details.
276 *
277 * - ndp_g_lock and ncec_lock for protecting NCEs.
278 *
279 * - ill_lock protects fields of the ill and ipif. Details in ip.h
280 *
281 * - ill_g_lock: This is a global reader/writer lock. Protects the following
282 *   * The AVL tree based global multi list of all ill's.
283 *   * The linked list of all ipifs of an ill
284 *   * The <ipsq-xop> mapping
285 *   * <ill-phyint> association
286 * Insertion/deletion of an ill in the system, insertion/deletion of an ipif
287 * into an ill, changing the <ipsq-xop> mapping of an ill, changing the
288 * <ill-phyint> assoc of an ill will all have to hold the ill_g_lock as
289 * writer for the actual duration of the insertion/deletion/change.
290 *
291 * - ill_lock: This is a per ill mutex.
292 * It protects some members of the ill_t struct; see ip.h for details.
293 * It also protects the <ill-phyint> assoc.
294 * It also protects the list of ipifs hanging off the ill.
295 *
296 * - ipsq_lock: This is a per ipsq_t mutex lock.
297 * This protects some members of the ipsq_t struct; see ip.h for details.
298 * It also protects the <ipsq-ipxop> mapping
299 *
300 * - ipx_lock: This is a per ipxop_t mutex lock.
301 * This protects some members of the ipxop_t struct; see ip.h for details.
302 *
303 * - phyint_lock: This is a per phyint mutex lock. Protects just the
304 *   phyint_flags
305 *
306 * - ip_addr_avail_lock: This is used to ensure the uniqueness of IP addresses.
307 * This lock is held in ipif_up_done and the ipif is marked IPIF_UP and the
308 * uniqueness check also done atomically.
309 *
310 * - ill_g_usersrc_lock: This readers/writer lock protects the usersrc
311 *   group list linked by ill_usersrc_grp_next. It also protects the
312 *   ill_usersrc_ifindex field. It is taken as a writer when a member of the
313 *   group is being added or deleted. This lock is taken as a reader when
314 *   walking the list/group(eg: to get the number of members in a usersrc group).
315 * Note, it is only necessary to take this lock if the ill_usersrc_grp_next
316 * field is changing state i.e from NULL to non-NULL or vice-versa. For
317 * example, it is not necessary to take this lock in the initial portion
318 * of ip_sioctl_slifusesrc or at all in ip_sioctl_flags since these
319 * operations are executed exclusively and that ensures that the "usersrc
320 * group state" cannot change. The "usersrc group state" change can happen
321 * only in the latter part of ip_sioctl_slifusesrc and in ill_delete.
322 *
323 * Changing <ill-phyint>, <ipsq-xop> associations:
324 *
325 * To change the <ill-phyint> association, the ill_g_lock must be held

```

```

326 * as writer, and the ill_locks of both the v4 and v6 instance of the ill
327 * must be held.
328 *
329 * To change the <ipsq-xop> association, the ill_g_lock must be held as
330 * writer, the ipsq_lock must be held, and one must be writer on the ipsq.
331 * This is only done when ill's are added or removed from IPMP groups.
332 *
333 * To add or delete an ipif from the list of ipifs hanging off the ill,
334 * ill_g_lock (writer) and ill_lock must be held and the thread must be
335 * a writer on the associated ipsq.
336 *
337 * To add or delete an ill to the system, the ill_g_lock must be held as
338 * writer and the thread must be a writer on the associated ipsq.
339 *
340 * To add or delete an ilm to an ill, the ill_lock must be held and the thread
341 * must be a writer on the associated ipsq.
342 *
343 * Lock hierarchy
344 *
345 * Some lock hierarchy scenarios are listed below.
346 *
347 * ill_g_lock -> conn_lock -> ill_lock -> ipsq_lock -> ipx_lock
348 * ill_g_lock -> ill_lock(s) -> phyint_lock
349 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock
350 * ill_g_lock -> ip_addr_avail_lock
351 * conn_lock -> irb_lock -> ill_lock -> ire_lock
352 * ill_g_lock -> ip_g_nd_lock
353 * ill_g_lock -> ips_ipmp_lock -> ill_lock -> nce_lock
354 * ill_g_lock -> ndp_g_lock -> ill_lock -> ncec_lock -> nce_lock
355 * arl_lock -> ill_lock
356 * ips_ire_dep_lock -> irb_lock
357 *
358 * When more than 1 ill lock is needed to be held, all ill lock addresses
359 * are sorted on address and locked starting from highest addressed lock
360 * downward.
361 *
362 * Multicast scenarios
363 * ips_ill_g_lock -> ill_mcast_lock
364 * conn_ilg_lock -> ips_ill_g_lock -> ill_lock
365 * ill_mcast_serializer -> ill_mcast_lock -> ips_ipmp_lock -> ill_lock
366 * ill_mcast_serializer -> ill_mcast_lock -> connnf_lock -> conn_lock
367 * ill_mcast_serializer -> ill_mcast_lock -> conn_ilg_lock
368 * ill_mcast_serializer -> ill_mcast_lock -> ips_igmp_timer_lock
369 *
370 * IPsec scenarios
371 *
372 * ipsa_lock -> ill_g_lock -> ill_lock
373 * ill_g_usersrc_lock -> ill_g_lock -> ill_lock
374 *
375 * Trusted Solaris scenarios
376 *
377 * igsa_lock -> gcgrp_rwlock -> gcgrp_lock
378 * igsa_lock -> gcdb_lock
379 * gcgrp_rwlock -> ire_lock
380 * gcgrp_rwlock -> gcdb_lock
381 *
382 * squeue(sq_lock), flow related (ft_lock, fe_lock) locking
383 *
384 * cpu_lock --> ill_lock --> sqset_lock --> sq_lock
385 * sq_lock -> conn_lock -> QLOCK(q)
386 * ill_lock -> ft_lock -> fe_lock
387 *
388 * Routing/forwarding table locking notes:
389 *
390 * Lock acquisition order: Radix tree lock, irb_lock.
391 * Requirements:

```

```

392 * i. Walker must not hold any locks during the walker callback.
393 * ii Walker must not see a truncated tree during the walk because of any node
394 * deletion.
395 * iii Existing code assumes ire_bucket is valid if it is non-null and is used
396 * in many places in the code to walk the irb list. Thus even if all the
397 * ires in a bucket have been deleted, we still can't free the radix node
398 * until the ires have actually been inactive'd (freed).
399 *
400 * Tree traversal - Need to hold the global tree lock in read mode.
401 * Before dropping the global tree lock, need to either increment the ire_refcnt
402 * to ensure that the radix node can't be deleted.
403 *
404 * Tree add - Need to hold the global tree lock in write mode to add a
405 * radix node. To prevent the node from being deleted, increment the
406 * irb_refcnt, after the node is added to the tree. The ire itself is
407 * added later while holding the irb_lock, but not the tree lock.
408 *
409 * Tree delete - Need to hold the global tree lock and irb_lock in write mode.
410 * All associated ires must be inactive (i.e. freed), and irb_refcnt
411 * must be zero.
412 *
413 * Walker - Increment irb_refcnt before calling the walker callback. Hold the
414 * global tree lock (read mode) for traversal.
415 *
416 * IRE dependencies - In some cases we hold ips_ire_dep_lock across ire_refrele
417 * hence we will acquire irb_lock while holding ips_ire_dep_lock.
418 *
419 * IPsec notes :
420 *
421 * IP interacts with the IPsec code (AH/ESP) by storing IPsec attributes
422 * in the ip_xmit_attr_t ip_rcv_attr_t. For outbound datagrams, the
423 * ip_xmit_attr_t has the
424 * information used by the IPsec code for applying the right level of
425 * protection. The information initialized by IP in the ip_xmit_attr_t
426 * is determined by the per-socket policy or global policy in the system.
427 * For inbound datagrams, the ip_rcv_attr_t
428 * starts out with nothing in it. It gets filled
429 * with the right information if it goes through the AH/ESP code, which
430 * happens if the incoming packet is secure. The information initialized
431 * by AH/ESP, is later used by IP (during fanouts to ULP) to see whether
432 * the policy requirements needed by per-socket policy or global policy
433 * is met or not.
434 *
435 * For fully connected sockets i.e dst, src [addr, port] is known,
436 * conn_policy_cached is set indicating that policy has been cached.
437 * conn_in_enforce_policy may or may not be set depending on whether
438 * there is a global policy match or per-socket policy match.
439 * Policy inheriting happens in ip_policy_set once the destination is known.
440 * Once the right policy is set on the conn_t, policy cannot change for
441 * this socket. This makes life simpler for TCP (UDP ?) where
442 * re-transmissions go out with the same policy. For symmetry, policy
443 * is cached for fully connected UDP sockets also. Thus if policy is cached,
444 * it also implies that policy is latched i.e policy cannot change
445 * on these sockets. As we have the right policy on the conn, we don't
446 * have to lookup global policy for every outbound and inbound datagram
447 * and thus serving as an optimization. Note that a global policy change
448 * does not affect fully connected sockets if they have policy. If fully
449 * connected sockets did not have any policy associated with it, global
450 * policy change may affect them.
451 *
452 * IP Flow control notes:
453 * -----
454 * Non-TCP streams are flow controlled by IP. The way this is accomplished
455 * differs when ILL_CAPAB_DLD_DIRECT is enabled for that IP instance. When
456 * ILL_DIRECT_CAPABLE(ill) is TRUE, IP can do direct function calls into
457 * GLDv3. Otherwise packets are sent down to lower layers using STREAMS

```

```

458 * functions.
459 *
460 * Per Tx ring udp flow control:
461 * This is applicable only when ILL_CAPAB_DLD_DIRECT capability is set in
462 * the ill (i.e. ILL_DIRECT_CAPABLE(ill) is true).
463 *
464 * The underlying link can expose multiple Tx rings to the GLDv3 mac layer.
465 * To achieve best performance, outgoing traffic need to be fanned out among
466 * these Tx ring. mac_tx() is called (via str_mdata_fastpath_put()) to send
467 * traffic out of the NIC and it takes a fanout hint. UDP connections pass
468 * the address of connp as fanout hint to mac_tx(). Under flow controlled
469 * condition, mac_tx() returns a non-NULL cookie (ip_mac_tx_cookie_t). This
470 * cookie points to a specific Tx ring that is blocked. The cookie is used to
471 * hash into an idl_tx_list[] entry in idl_tx_list[] array. Each idl_tx_list_t
472 * point to drain_lists (idl_t's). These drain list will store the blocked UDP
473 * connp's. The drain list is not a single list but a configurable number of
474 * lists.
475 *
476 * The diagram below shows idl_tx_list_t's and their drain_lists. ip_stack_t
477 * has an array of idl_tx_list_t. The size of the array is TX_FANOUT_SIZE
478 * which is equal to 128. This array in turn contains a pointer to idl_t[],
479 * the ip drain list. The idl_t[] array size is MIN(max_ncpus, 8). The drain
480 * list will point to the list of connp's that are flow controlled.
481 *
482 *
483 *
484 *
485 *
486 *
487 *
488 * |idl_tx_list[0]|->
489 * -----
490 *
491 *
492 *
493 *
494 *
495 *
496 *
497 *
498 *
499 * -----
500 * |idl_tx_list[1]|->
501 * -----
502 *
503 *
504 *
505 *
506 *
507 *
508 * -----
509 * |idl_tx_list[n]|-> ...
510 * -----
511 *
512 * When mac_tx() returns a cookie, the cookie is hashed into an index into
513 * ips_idl_tx_list[], and conn_drain_insert() is called with the idl_tx_list
514 * to insert the conn onto. conn_drain_insert() asserts flow control for the
515 * sockets via su_txq_full() (non-STREAMS) or QFULL on conn_wq (STREAMS).
516 * Further, conn_blocked is set to indicate that the conn is blocked.
517 *
518 * GLDv3 calls ill_flow_enable() when flow control is relieved. The cookie
519 * passed in the call to ill_flow_enable() identifies the blocked Tx ring and
520 * is again hashed to locate the appropriate idl_tx_list, which is then
521 * drained via conn_walk_drain(). conn_walk_drain() goes through each conn in
522 * the drain list and calls conn_drain_remove() to clear flow control (via
523 * calling su_txq_full() or clearing QFULL), and remove the conn from the

```

```

524 * Note that the drain list is not a single list but a (configurable) array of
525 * lists (8 elements by default). Synchronization between drain insertion and
526 * flow control wakeup is handled by using idl_txl->txl_lock, and only
527 * conn_drain_insert() and conn_drain_remove() manipulate the drain list.
528 *
529 * Flow control via STREAMS is used when ILL_DIRECT_CAPABLE() returns FALSE.
530 * On the send side, if the packet cannot be sent down to the driver by IP
531 * (canput() fails), ip_xmit() drops the packet and returns EWOULDBLOCK to the
532 * caller, who may then invoke ixa_check_drain_insert() to insert the conn on
533 * the 0'th drain list. When ip_wsrv() runs on the ill_wq because flow
534 * control has been relieved, the blocked conns in the 0'th drain list are
535 * drained as in the non-STREAMS case.
536 *
537 * In both the STREAMS and non-STREAMS cases, the sockfs upcall to set QFULL
538 * is done when the conn is inserted into the drain list (conn_drain_insert())
539 * and cleared when the conn is removed from the it (conn_drain_remove()).
540 *
541 * IPQoS notes:
542 *
543 * IPQoS Policies are applied to packets using IPPF (IP Policy framework)
544 * and IPQoS modules. IPPF includes hooks in IP at different control points
545 * (callout positions) which direct packets to IPQoS modules for policy
546 * processing. Policies, if present, are global.
547 *
548 * The callout positions are located in the following paths:
549 *     o local_in (packets destined for this host)
550 *     o local_out (packets originating from this host)
551 *     o fwd_in (packets forwarded by this m/c - inbound)
552 *     o fwd_out (packets forwarded by this m/c - outbound)
553 * Hooks at these callout points can be enabled/disabled using the ndd variable
554 * ip_policy_mask (a bit mask with the 4 LSB indicating the callout positions).
555 * By default all the callout positions are enabled.
556 *
557 * Outbound (local_out)
558 * Hooks are placed in ire_send_wire_v4 and ire_send_wire_v6.
559 *
560 * Inbound (local_in)
561 * Hooks are placed in ip_fanout_v4 and ip_fanout_v6.
562 *
563 * Forwarding (in and out)
564 * Hooks are placed in ire_rcv_forward_v4/v6.
565 *
566 * IP Policy Framework processing (IPPF processing)
567 * Policy processing for a packet is initiated by ip_process, which ascertains
568 * that the classifier (ipgpc) is loaded and configured, failing which the
569 * packet resumes normal processing in IP. If the classifier is present, the
570 * packet is acted upon by one or more IPQoS modules (action instances), per
571 * filters configured in ipgpc and resumes normal IP processing thereafter.
572 * An action instance can drop a packet in course of its processing.
573 *
574 * Zones notes:
575 *
576 * The partitioning rules for networking are as follows:
577 * 1) Packets coming from a zone must have a source address belonging to that
578 *    zone.
579 * 2) Packets coming from a zone can only be sent on a physical interface on
580 *    which the zone has an IP address.
581 * 3) Between two zones on the same machine, packet delivery is only allowed if
582 *    there's a matching route for the destination and zone in the forwarding
583 *    table.
584 * 4) The TCP and UDP port spaces are per-zone; that is, two processes in
585 *    different zones can bind to the same port with the wildcard address
586 *    (INADDR_ANY).
587 *
588 * The granularity of interface partitioning is at the logical interface level.
589 * Therefore, every zone has its own IP addresses, and incoming packets can be

```

```

590 * attributed to a zone unambiguously. A logical interface is placed into a zone
591 * using the SIOCSLIFZONE ioctl; this sets the ipif_zoneid field in the ipif_t
592 * structure. Rule (1) is implemented by modifying the source address selection
593 * algorithm so that the list of eligible addresses is filtered based on the
594 * sending process zone.
595 *
596 * The Internet Routing Entries (IREs) are either exclusive to a zone or shared
597 * across all zones, depending on their type. Here is the break-up:
598 *
599 * IRE type                                Shared/exclusive
600 * -----
601 * IRE_BROADCAST                           Exclusive
602 * IRE_DEFAULT (default routes)            Shared (*)
603 * IRE_LOCAL                               Exclusive (x)
604 * IRE_LOOPBACK                            Exclusive
605 * IRE_PREFIX (net routes)                 Shared (*)
606 * IRE_IF_NORESOLVER (interface routes)    Exclusive
607 * IRE_IF_RESOLVER (interface routes)       Exclusive
608 * IRE_IF_CLONE (interface routes)          Exclusive
609 * IRE_HOST (host routes)                  Shared (*)
610 *
611 * (*) A zone can only use a default or off-subnet route if the gateway is
612 * directly reachable from the zone, that is, if the gateway's address matches
613 * one of the zone's logical interfaces.
614 *
615 * (x) IRE_LOCAL are handled a bit differently.
616 * When ip_restrict_interzone_loopback is set (the default),
617 * ire_route_recursive restricts loopback using an IRE_LOCAL
618 * between zone to the case when L2 would have conceptually looped the packet
619 * back, i.e. the loopback which is required since neither Ethernet drivers
620 * nor Ethernet hardware loops them back. This is the case when the normal
621 * routes (ignoring IREs with different zoneids) would send out the packet on
622 * the same ill as the ill with which is IRE_LOCAL is associated.
623 *
624 * Multiple zones can share a common broadcast address; typically all zones
625 * share the 255.255.255.255 address. Incoming as well as locally originated
626 * broadcast packets must be dispatched to all the zones on the broadcast
627 * network. For directed broadcasts (e.g. 10.16.72.255) this is not trivial
628 * since some zones may not be on the 10.16.72/24 network. To handle this, each
629 * zone has its own set of IRE_BROADCAST entries; then, broadcast packets are
630 * sent to every zone that has an IRE_BROADCAST entry for the destination
631 * address on the input ill, see ip_input_broadcast().
632 *
633 * Applications in different zones can join the same multicast group address.
634 * The same logic applies for multicast as for broadcast. ip_input_multicast
635 * dispatches packets to all zones that have members on the physical interface.
636 */
637
638 /*
639 * Squeue Fanout flags:
640 *     0: No fanout.
641 *     1: Fanout across all squeues
642 */
643 boolean_t    ip_squeue_fanout = 0;
644
645 /*
646 * Maximum dups allowed per packet.
647 */
648 uint_t ip_max_frag_dups = 10;
649
650 static int    ip_open(queue_t *q, dev_t *devp, int flag, int sflag,
651                      cred_t *credp, boolean_t isv6);
652 static mblk_t *ip_xmit_attach_llhdr(mblk_t *, nce_t *);
653
654 static boolean_t icmp_inbound_verify_v4(mblk_t *, icmp_h_t *, ip_rcv_attr_t *);
655 static void     icmp_inbound_too_big_v4(icmp_h_t *, ip_rcv_attr_t *);

```

```

656 static void icmp_inbound_error_fanout_v4(mblk_t *, icmph_t *,
657 ip_rcv_attr_t *);
658 static void icmp_options_update(ipha_t *);
659 static void icmp_param_problem(mblk_t *, uint8_t, ip_rcv_attr_t *);
660 static void icmp_pkt(mblk_t *, void *, size_t, ip_rcv_attr_t *);
661 static mblk_t *icmp_pkt_err_ok(mblk_t *, ip_rcv_attr_t *);
662 static void icmp_redirect_v4(mblk_t *mp, ipha_t *, icmph_t *,
663 ip_rcv_attr_t *);
664 static void icmp_send_redirect(mblk_t *, ipaddr_t, ip_rcv_attr_t *);
665 static void icmp_send_reply_v4(mblk_t *, ipha_t *, icmph_t *,
666 ip_rcv_attr_t *);

668 mblk_t *ip_dlpi_alloc(size_t, t_uscalar_t);
669 char *ip_dot_addr(ipaddr_t, char *);
670 mblk_t *ip_carve_mp(mblk_t **, ssize_t);
671 int ip_close(queue_t *, int);
672 static char *ip_dot_saddr(uchar_t *, char *);
673 static void ip_lrput(queue_t *, mblk_t *);
674 ipaddr_t ip_net_mask(ipaddr_t);
675 char *ip_nv_lookup(nv_t *, int);
676 void ip_rput(queue_t *, mblk_t *);
677 static void ip_rput_dlpi_writer(ipsq_t *dummy_sq, queue_t *q, mblk_t *mp,
678 void *dummy_arg);
679 int ip_snmp_get(queue_t *, mblk_t *, int, boolean_t);
680 static mblk_t *ip_snmp_get_mib2_ip(queue_t *, mblk_t *,
681 mib2_ipIfStatsEntry_t *, ip_stack_t *, boolean_t);
682 static mblk_t *ip_snmp_get_mib2_ip_traffic_stats(queue_t *, mblk_t *,
683 ip_stack_t *, boolean_t);
684 static mblk_t *ip_snmp_get_mib2_ip6(queue_t *, mblk_t *, ip_stack_t *,
685 boolean_t);
686 static mblk_t *ip_snmp_get_mib2_icmp(queue_t *, mblk_t *, ip_stack_t *ipst);
687 static mblk_t *ip_snmp_get_mib2_icmp6(queue_t *, mblk_t *, ip_stack_t *ipst);
688 static mblk_t *ip_snmp_get_mib2_igmp(queue_t *, mblk_t *, ip_stack_t *ipst);
689 static mblk_t *ip_snmp_get_mib2_multi(queue_t *, mblk_t *, ip_stack_t *ipst);
690 static mblk_t *ip_snmp_get_mib2_ip_addr(queue_t *, mblk_t *,
691 ip_stack_t *ipst, boolean_t);
692 static mblk_t *ip_snmp_get_mib2_ip6_addr(queue_t *, mblk_t *,
693 ip_stack_t *ipst, boolean_t);
694 static mblk_t *ip_snmp_get_mib2_ip_group_src(queue_t *, mblk_t *,
695 ip_stack_t *ipst);
696 static mblk_t *ip_snmp_get_mib2_ip6_group_src(queue_t *, mblk_t *,
697 ip_stack_t *ipst);
698 static mblk_t *ip_snmp_get_mib2_ip_group_mem(queue_t *, mblk_t *,
699 ip_stack_t *ipst);
700 static mblk_t *ip_snmp_get_mib2_ip6_group_mem(queue_t *, mblk_t *,
701 ip_stack_t *ipst);
702 static mblk_t *ip_snmp_get_mib2_virt_multi(queue_t *, mblk_t *,
703 ip_stack_t *ipst);
704 static mblk_t *ip_snmp_get_mib2_multi_rtable(queue_t *, mblk_t *,
705 ip_stack_t *ipst);
706 static mblk_t *ip_snmp_get_mib2_ip_route_media(queue_t *, mblk_t *, int,
707 ip_stack_t *ipst);
708 static mblk_t *ip_snmp_get_mib2_ip6_route_media(queue_t *, mblk_t *, int,
709 ip_stack_t *ipst);
710 static void ip_snmp_get2_v4(ire_t *, iproudata_t *);
711 static void ip_snmp_get2_v6_route(ire_t *, iproudata_t *);
712 static int ip_snmp_get2_v4_media(ncec_t *, iproudata_t *);
713 static int ip_snmp_get2_v6_media(ncec_t *, iproudata_t *);
714 int ip_snmp_set(queue_t *, int, int, uchar_t *, int);

716 static mblk_t *ip_fragment_copyhdr(uchar_t *, int, int, ip_stack_t *,
717 mblk_t *);

719 static void conn_drain_init(ip_stack_t *);
720 static void conn_drain_fini(ip_stack_t *);
721 static void conn_drain(conn_t *connp, boolean_t closing);

```

```

723 static void conn_walk_drain(ip_stack_t *, idl_tx_list_t *);
724 static void conn_walk_sctp(pfv_t, void *, zoneid_t, netstack_t *);

726 static void *ip_stack_init(netstackid_t stackid, netstack_t *ns);
727 static void ip_stack_shutdown(netstackid_t stackid, void *arg);
728 static void ip_stack_fini(netstackid_t stackid, void *arg);

730 static int ip_multirtt_apply_membership(int (*fn)(conn_t *, boolean_t,
731 const in6_addr_t *, ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *),
732 ire_t *, conn_t *, boolean_t, const in6_addr_t *, mcast_record_t,
733 const in6_addr_t *);

735 static int ip_queue_switch(int);

737 static void *ip_kstat_init(netstackid_t, ip_stack_t *);
738 static void ip_kstat_fini(netstackid_t, kstat_t *);
739 static int ip_kstat_update(kstat_t *kp, int rw);
740 static void *icmp_kstat_init(netstackid_t);
741 static void icmp_kstat_fini(netstackid_t, kstat_t *);
742 static int icmp_kstat_update(kstat_t *kp, int rw);
743 static void *ip_kstat2_init(netstackid_t, ip_stat_t *);
744 static void ip_kstat2_fini(netstackid_t, kstat_t *);

746 static void ipobs_init(ip_stack_t *);
747 static void ipobs_fini(ip_stack_t *);

749 static int ip_tp_cpu_update(cpu_setup_t, int, void *);

751 ipaddr_t ip_g_all_ones = IP_HOST_MASK;

753 static long ip_rput_pullups;
754 int dohwcksum = 1; /* use h/w cksum if supported by the hardware */

756 vmem_t *ip_minor_arena_sa; /* for minor nos. from INET_MIN_DEV+2 thru 2^18-1 */
757 vmem_t *ip_minor_arena_la; /* for minor nos. from 2^18 thru 2^32-1 */

759 int ip_debug;

761 /*
762 * Multirouting/CGTP stuff
763 */
764 int ip_cgtp_filter_rev = CGTP_FILTER_REV; /* CGTP hooks version */

766 /*
767 * IP tunables related declarations. Definitions are in ip_tunables.c
768 */
769 extern mod_prop_info_t ip_propinfo_tbl[];
770 extern int ip_propinfo_count;

772 /*
773 * Table of IP ioctls encoding the various properties of the ioctl and
774 * indexed based on the last byte of the ioctl command. Occasionally there
775 * is a clash, and there is more than 1 ioctl with the same last byte.
776 * In such a case 1 ioctl is encoded in the ndx table and the remaining
777 * ioctls are encoded in the misc table. An entry in the ndx table is
778 * retrieved by indexing on the last byte of the ioctl command and comparing
779 * the ioctl command with the value in the ndx table. In the event of a
780 * mismatch the misc table is then searched sequentially for the desired
781 * ioctl command.
782 *
783 * Entry: <command> <copyin_size> <flags> <cmd_type> <function> <restart_func>
784 */
785 ip_ioctl_cmd_t ip_ndx_ioctl_table[] = {
786 /* 000 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
787 /* 001 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

```

```

788 /* 002 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
789 /* 003 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
790 /* 004 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
791 /* 005 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
792 /* 006 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
793 /* 007 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
794 /* 008 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
795 /* 009 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

797 /* 010 */ { SIOCADDR, sizeof (struct rtentry), IPI_PRIV,
798             MISC_CMD, ip_siocaddr, NULL },
799 /* 011 */ { SIOCDELRT, sizeof (struct rtentry), IPI_PRIV,
800             MISC_CMD, ip_siocdelrt, NULL },

802 /* 012 */ { SIOCSIFADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
803             IF_CMD, ip_siocifl_addr, ip_siocifl_addr_restart },
804 /* 013 */ { SIOCGIFADDR, sizeof (struct ifreq), IPI_GET_CMD,
805             IF_CMD, ip_siocifl_get_addr, NULL },

807 /* 014 */ { SIOCSIFDSTADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
808             IF_CMD, ip_siocifl_dstaddr, ip_siocifl_dstaddr_restart },
809 /* 015 */ { SIOCGIFDSTADDR, sizeof (struct ifreq),
810             IPI_GET_CMD, IF_CMD, ip_siocifl_get_dstaddr, NULL },

812 /* 016 */ { SIOCSIFFLAGS, sizeof (struct ifreq),
813             IPI_PRIV | IPI_WR,
814             IF_CMD, ip_siocifl_flags, ip_siocifl_flags_restart },
815 /* 017 */ { SIOCGIFFLAGS, sizeof (struct ifreq),
816             IPI_MODOK | IPI_GET_CMD,
817             IF_CMD, ip_siocifl_get_flags, NULL },

819 /* 018 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
820 /* 019 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

822 /* copyin size cannot be coded for SIOCGIFCONF */
823 /* 020 */ { O_SIOCGIFCONF, 0, IPI_GET_CMD,
824             MISC_CMD, ip_siocifl_get_ifconf, NULL },

826 /* 021 */ { SIOCSIFMTU, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
827             IF_CMD, ip_siocifl_mtu, NULL },
828 /* 022 */ { SIOCGIFMTU, sizeof (struct ifreq), IPI_GET_CMD,
829             IF_CMD, ip_siocifl_get_mtu, NULL },
830 /* 023 */ { SIOCSIFBRDADDR, sizeof (struct ifreq),
831             IPI_GET_CMD, IF_CMD, ip_siocifl_get_brddaddr, NULL },
832 /* 024 */ { SIOCSIFBRDADDR, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
833             IF_CMD, ip_siocifl_brddaddr, NULL },
834 /* 025 */ { SIOCGIFNETMASK, sizeof (struct ifreq),
835             IPI_GET_CMD, IF_CMD, ip_siocifl_get_netmask, NULL },
836 /* 026 */ { SIOCSIFNETMASK, sizeof (struct ifreq), IPI_PRIV | IPI_WR,
837             IF_CMD, ip_siocifl_netmask, ip_siocifl_netmask_restart },
838 /* 027 */ { SIOCGIFMETRIC, sizeof (struct ifreq),
839             IPI_GET_CMD, IF_CMD, ip_siocifl_get_metric, NULL },
840 /* 028 */ { SIOCSIFMETRIC, sizeof (struct ifreq), IPI_PRIV,
841             IF_CMD, ip_siocifl_metric, NULL },
842 /* 029 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

844 /* See 166-168 below for extended SIOC*ARP ioctls */
845 /* 030 */ { SIOCSARP, sizeof (struct arpreq), IPI_PRIV | IPI_WR,
846             ARP_CMD, ip_siocifl_arp, NULL },
847 /* 031 */ { SIOCGARP, sizeof (struct arpreq), IPI_GET_CMD,
848             ARP_CMD, ip_siocifl_arp, NULL },
849 /* 032 */ { SIOCDAARP, sizeof (struct arpreq), IPI_PRIV | IPI_WR,
850             ARP_CMD, ip_siocifl_arp, NULL },

852 /* 033 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
853 /* 034 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

```

```

854 /* 035 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
855 /* 036 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
856 /* 037 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
857 /* 038 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
858 /* 039 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
859 /* 040 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
860 /* 041 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
861 /* 042 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
862 /* 043 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
863 /* 044 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
864 /* 045 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
865 /* 046 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
866 /* 047 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
867 /* 048 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
868 /* 049 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
869 /* 050 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
870 /* 051 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
871 /* 052 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
872 /* 053 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

874 /* 054 */ { IF_UNITSEL, sizeof (int), IPI_PRIV | IPI_WR | IPI_MODOK,
875             MISC_CMD, if_unitssel, if_unitssel_restart },

877 /* 055 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
878 /* 056 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
879 /* 057 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
880 /* 058 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
881 /* 059 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
882 /* 060 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
883 /* 061 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
884 /* 062 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
885 /* 063 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
886 /* 064 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
887 /* 065 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
888 /* 066 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
889 /* 067 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
890 /* 068 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
891 /* 069 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
892 /* 070 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
893 /* 071 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
894 /* 072 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

896 /* 073 */ { SIOCSIFNAME, sizeof (struct ifreq),
897             IPI_PRIV | IPI_WR | IPI_MODOK,
898             IF_CMD, ip_siocifl_sifname, NULL },

900 /* 074 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
901 /* 075 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
902 /* 076 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
903 /* 077 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
904 /* 078 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
905 /* 079 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
906 /* 080 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
907 /* 081 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
908 /* 082 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
909 /* 083 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
910 /* 084 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
911 /* 085 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
912 /* 086 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

914 /* 087 */ { SIOCGIFNUM, sizeof (int), IPI_GET_CMD,
915             MISC_CMD, ip_siocifl_get_ifnum, NULL },
916 /* 088 */ { SIOCGIFMUXID, sizeof (struct ifreq), IPI_GET_CMD,
917             IF_CMD, ip_siocifl_get_muxid, NULL },
918 /* 089 */ { SIOCSIFMUXID, sizeof (struct ifreq),
919             IPI_PRIV | IPI_WR, IF_CMD, ip_siocifl_muxid, NULL },

```



```

921 /* Both if and lif variants share same func */
922 /* 090 */ { SIOCGIFINDEX, sizeof (struct lifreq), IPI_GET_CMD,
923             IF_CMD, ip_siocctl_get_lifindex, NULL },
924 /* Both if and lif variants share same func */
925 /* 091 */ { SIOCSIFINDEX, sizeof (struct lifreq),
926             IPI_PRIV | IPI_WR, IF_CMD, ip_siocctl_slifindex, NULL },

928 /* copyin size cannot be coded for SIOCGIFCONF */
929 /* 092 */ { SIOCGIFCONF, 0, IPI_GET_CMD,
930             MISC_CMD, ip_siocctl_get_ifconf, NULL },
931 /* 093 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
932 /* 094 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
933 /* 095 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
934 /* 096 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
935 /* 097 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
936 /* 098 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
937 /* 099 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
938 /* 100 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
939 /* 101 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
940 /* 102 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
941 /* 103 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
942 /* 104 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
943 /* 105 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
944 /* 106 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
945 /* 107 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
946 /* 108 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
947 /* 109 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

949 /* 110 */ { SIOCLIFREMOVEIF, sizeof (struct lifreq),
950             IPI_PRIV | IPI_WR, LIF_CMD, ip_siocctl_removeif,
951             ip_siocctl_removeif_restart },
952 /* 111 */ { SIOCLIFADDIF, sizeof (struct lifreq),
953             IPI_GET_CMD | IPI_PRIV | IPI_WR,
954             LIF_CMD, ip_siocctl_addif, NULL },
955 #define SIOCLIFADDR_NDX 112
956 /* 112 */ { SIOCLIFADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
957             LIF_CMD, ip_siocctl_addr, ip_siocctl_addr_restart },
958 /* 113 */ { SIOCLIFADDR, sizeof (struct lifreq),
959             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_addr, NULL },
960 /* 114 */ { SIOCSLIFDSTADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
961             LIF_CMD, ip_siocctl_dstaddr, ip_siocctl_dstaddr_restart },
962 /* 115 */ { SIOCLIFDSTADDR, sizeof (struct lifreq),
963             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_dstaddr, NULL },
964 /* 116 */ { SIOCSLIFFLAGS, sizeof (struct lifreq),
965             IPI_PRIV | IPI_WR,
966             LIF_CMD, ip_siocctl_flags, ip_siocctl_flags_restart },
967 /* 117 */ { SIOCLIFFLAGS, sizeof (struct lifreq),
968             IPI_GET_CMD | IPI_MODOK,
969             LIF_CMD, ip_siocctl_get_flags, NULL },

971 /* 118 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
972 /* 119 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

974 /* 120 */ { O_SIOCLIFCONF, 0, IPI_GET_CMD, MISC_CMD,
975             ip_siocctl_get_lifconf, NULL },
976 /* 121 */ { SIOCLIFMTU, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
977             LIF_CMD, ip_siocctl_mtu, NULL },
978 /* 122 */ { SIOCLIFMTU, sizeof (struct lifreq), IPI_GET_CMD,
979             LIF_CMD, ip_siocctl_get_mtu, NULL },
980 /* 123 */ { SIOCLIFBRDADDR, sizeof (struct lifreq),
981             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_brddaddr, NULL },
982 /* 124 */ { SIOCSLIFBRDADDR, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
983             LIF_CMD, ip_siocctl_brddaddr, NULL },
984 /* 125 */ { SIOCLIFNETMASK, sizeof (struct lifreq),
985             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_netmask, NULL },

```

```

986 /* 126 */ { SIOCLIFNETMASK, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
987             LIF_CMD, ip_siocctl_netmask, ip_siocctl_netmask_restart },
988 /* 127 */ { SIOCLIFMETRIC, sizeof (struct lifreq),
989             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_metric, NULL },
990 /* 128 */ { SIOCSLIFMETRIC, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
991             LIF_CMD, ip_siocctl_metric, NULL },
992 /* 129 */ { SIOCSLIFNAME, sizeof (struct lifreq),
993             IPI_PRIV | IPI_WR | IPI_MODOK,
994             LIF_CMD, ip_siocctl_slifname,
995             ip_siocctl_slifname_restart },

997 /* 130 */ { SIOCLIFNUM, sizeof (struct lifnum), IPI_GET_CMD,
998             MISC_CMD, ip_siocctl_get_lifnum, NULL },
999 /* 131 */ { SIOCLIFMUXID, sizeof (struct lifreq),
1000             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_muxid, NULL },
1001 /* 132 */ { SIOCSLIFMUXID, sizeof (struct lifreq),
1002             IPI_PRIV | IPI_WR, LIF_CMD, ip_siocctl_muxid, NULL },
1003 /* 133 */ { SIOCLIFINDEX, sizeof (struct lifreq),
1004             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_lifindex, 0 },
1005 /* 134 */ { SIOCSLIFINDEX, sizeof (struct lifreq),
1006             IPI_PRIV | IPI_WR, LIF_CMD, ip_siocctl_slifindex, 0 },
1007 /* 135 */ { SIOCSLIFTOKEN, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1008             LIF_CMD, ip_siocctl_token, NULL },
1009 /* 136 */ { SIOCLIFTOKEN, sizeof (struct lifreq),
1010             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_token, NULL },
1011 /* 137 */ { SIOCSLIFSUBNET, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1012             LIF_CMD, ip_siocctl_subnet, ip_siocctl_subnet_restart },
1013 /* 138 */ { SIOCLIFSUBNET, sizeof (struct lifreq),
1014             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_subnet, NULL },
1015 /* 139 */ { SIOCSLIFLNKINFO, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1016             LIF_CMD, ip_siocctl_lnkinfo, NULL },

1018 /* 140 */ { SIOCLIFLNKINFO, sizeof (struct lifreq),
1019             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_lnkinfo, NULL },
1020 /* 141 */ { SIOCLIFDELND, sizeof (struct lifreq), IPI_PRIV,
1021             LIF_CMD, ip_siocdelndp_v6, NULL },
1022 /* 142 */ { SIOCLIFGETND, sizeof (struct lifreq), IPI_GET_CMD,
1023             LIF_CMD, ip_siocqueryndp_v6, NULL },
1024 /* 143 */ { SIOCLIFSETND, sizeof (struct lifreq), IPI_PRIV,
1025             LIF_CMD, ip_siocsetndp_v6, NULL },
1026 /* 144 */ { SIOCTMYADDR, sizeof (struct sioc_addrreq), IPI_GET_CMD,
1027             MISC_CMD, ip_siocctl_tmyaddr, NULL },
1028 /* 145 */ { SIOCTONLINK, sizeof (struct sioc_addrreq), IPI_GET_CMD,
1029             MISC_CMD, ip_siocctl_tonlink, NULL },
1030 /* 146 */ { SIOCTMYSITE, sizeof (struct sioc_addrreq), 0,
1031             MISC_CMD, ip_siocctl_tmysite, NULL },
1032 /* 147 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1033 /* 148 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1034 /* IPSECioctls handled in ip_siocctl_copyin_setup itself */
1035 /* 149 */ { SIOCFIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1036 /* 150 */ { SIOCSIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1037 /* 151 */ { SIOCDIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },
1038 /* 152 */ { SIOCLIPSECONFIG, 0, IPI_PRIV, MISC_CMD, NULL, NULL },

1040 /* 153 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1042 /* 154 */ { SIOCLIFBINDING, sizeof (struct lifreq), IPI_GET_CMD,
1043             LIF_CMD, ip_siocctl_get_binding, NULL },
1044 /* 155 */ { SIOCSLIFGROUPNAME, sizeof (struct lifreq),
1045             IPI_PRIV | IPI_WR,
1046             LIF_CMD, ip_siocctl_groupname, ip_siocctl_groupname },
1047 /* 156 */ { SIOCLIFGROUPNAME, sizeof (struct lifreq),
1048             IPI_GET_CMD, LIF_CMD, ip_siocctl_get_groupname, NULL },
1049 /* 157 */ { SIOCLIFGROUPINFO, sizeof (lifgroupinfo_t),
1050             IPI_GET_CMD, MISC_CMD, ip_siocctl_groupinfo, NULL },

```

```

1052 /* Leave 158-160 unused; used to be SIOC*IFARP ioctls */
1053 /* 158 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1054 /* 159 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1055 /* 160 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1057 /* 161 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },

1059 /* These are handled in ip_siocctl_copyin_setup itself */
1060 /* 162 */ { SIOCGIP6ADDRPOLICY, 0, IPI_NULL_BCONT,
1061          MISC_CMD, NULL, NULL },
1062 /* 163 */ { SIOCSIP6ADDRPOLICY, 0, IPI_PRIV | IPI_NULL_BCONT,
1063          MISC_CMD, NULL, NULL },
1064 /* 164 */ { SIOCGDSTINFO, 0, IPI_GET_CMD, MISC_CMD, NULL, NULL },

1066 /* 165 */ { SIOCLIFCONF, 0, IPI_GET_CMD, MISC_CMD,
1067          ip_siocctl_get_lifconf, NULL },

1069 /* 166 */ { SIOCSXARP, sizeof (struct xarpreq), IPI_PRIV | IPI_WR,
1070          XARP_CMD, ip_siocctl_arp, NULL },
1071 /* 167 */ { SIOCGXARP, sizeof (struct xarpreq), IPI_GET_CMD,
1072          XARP_CMD, ip_siocctl_arp, NULL },
1073 /* 168 */ { SIOCDXARP, sizeof (struct xarpreq), IPI_PRIV | IPI_WR,
1074          XARP_CMD, ip_siocctl_arp, NULL },

1076 /* SIOCPPOCKFS is not handled by IP */
1077 /* 169 */ { IPI_DONTCARE /* SIOCPPOCKFS */, 0, 0, 0, NULL, NULL },

1079 /* 170 */ { SIOCLIFZONE, sizeof (struct lifreq),
1080          IPI_GET_CMD, LIF_CMD, ip_siocctl_get_lifzone, NULL },
1081 /* 171 */ { SIOCSLIFZONE, sizeof (struct lifreq),
1082          IPI_PRIV | IPI_WR, LIF_CMD, ip_siocctl_slifzone,
1083          ip_siocctl_slifzone_restart },
1084 /* 172-174 are SCTP ioctls and not handled by IP */
1085 /* 172 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1086 /* 173 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1087 /* 174 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1088 /* 175 */ { SIOCLIFUSESRC, sizeof (struct lifreq),
1089          IPI_GET_CMD, LIF_CMD,
1090          ip_siocctl_get_lifusesrc, 0 },
1091 /* 176 */ { SIOCSLIFUSESRC, sizeof (struct lifreq),
1092          IPI_PRIV | IPI_WR,
1093          LIF_CMD, ip_siocctl_slifusesrc,
1094          NULL },
1095 /* 177 */ { SIOCLIFSRCONF, 0, IPI_GET_CMD, MISC_CMD,
1096          ip_siocctl_get_lifsrcconf, NULL },
1097 /* 178 */ { SIOCGMSFILTER, sizeof (struct group_filter), IPI_GET_CMD,
1098          MSFILT_CMD, ip_siocctl_msfilter, NULL },
1099 /* 179 */ { SIOCSMSFILTER, sizeof (struct group_filter), 0,
1100          MSFILT_CMD, ip_siocctl_msfilter, NULL },
1101 /* 180 */ { SIOCGIPMSFILTER, sizeof (struct ip_msfilter), IPI_GET_CMD,
1102          MSFILT_CMD, ip_siocctl_msfilter, NULL },
1103 /* 181 */ { SIOCSIPMSFILTER, sizeof (struct ip_msfilter), 0,
1104          MSFILT_CMD, ip_siocctl_msfilter, NULL },
1105 /* 182 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
1106 /* SIOCSENABLESDP is handled by SDP */
1107 /* 183 */ { IPI_DONTCARE /* SIOCSENABLESDP */, 0, 0, 0, NULL, NULL },
1108 /* 184 */ { IPI_DONTCARE /* SIOCSQPTR */, 0, 0, 0, NULL, NULL },
1109 /* 185 */ { SIOCGIFHWADDR, sizeof (struct ifreq), IPI_GET_CMD,
1110          IF_CMD, ip_siocctl_get_ifhwaddr, NULL },
1111 /* 186 */ { IPI_DONTCARE /* SIOCGSTAMP */, 0, 0, 0, NULL, NULL },
1112 /* 187 */ { SIOCILB, 0, IPI_PRIV | IPI_GET_CMD, MISC_CMD,
1113          ip_siocctl_ilb_cmd, NULL },
1114 /* 188 */ { SIOCGPROP, 0, IPI_GET_CMD, 0, NULL, NULL },
1115 /* 189 */ { SIOCSPROP, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1116 /* 190 */ { SIOCLIFDADSTATE, sizeof (struct lifreq),
1117          IPI_GET_CMD, LIF_CMD, ip_siocctl_get_dadstate, NULL },

```

```

1118 /* 191 */ { SIOCSLIFPREFIX, sizeof (struct lifreq), IPI_PRIV | IPI_WR,
1119          LIF_CMD, ip_siocctl_prefix, ip_siocctl_prefix_restart },
1120 /* 192 */ { SIOCLIFHWADDR, sizeof (struct lifreq), IPI_GET_CMD,
1121          LIF_CMD, ip_siocctl_get_lifhwaddr, NULL }
1122 };

1124 int ip_ndx_ioctl_count = sizeof (ip_ndx_ioctl_table) / sizeof (ip_ioctl_cmd_t);

1126 ip_ioctl_cmd_t ip_misc_ioctl_table[] = {
1127     { I_LINK, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1128     { I_UNLINK, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1129     { I_PLINK, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1130     { I_PUNLINK, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1131     { ND_GET, 0, 0, 0, NULL, NULL },
1132     { ND_SET, 0, IPI_PRIV | IPI_WR, 0, NULL, NULL },
1133     { IP_IOCTL, 0, 0, 0, NULL, NULL },
1134     { SIOCGVIFCNT, sizeof (struct sioc_vif_req), IPI_GET_CMD,
1135          MISC_CMD, mrt_ioctl },
1136     { SIOCGTSGCNT, sizeof (struct sioc_sg_req), IPI_GET_CMD,
1137          MISC_CMD, mrt_ioctl },
1138     { SIOGETLSGCNT, sizeof (struct sioc_lsg_req), IPI_GET_CMD,
1139          MISC_CMD, mrt_ioctl }
1140 };

1142 int ip_misc_ioctl_count =
1143     sizeof (ip_misc_ioctl_table) / sizeof (ip_ioctl_cmd_t);

1145 int conn_drain_nthreads; /* Number of drainers reqd. */
1146 /* Settable in /etc/system */

1147 /* Defined in ip_ire.c */
1148 extern uint32_t ip_ire_max_bucket_cnt, ip6_ire_max_bucket_cnt;
1149 extern uint32_t ip_ire_min_bucket_cnt, ip6_ire_min_bucket_cnt;
1150 extern uint32_t ip_ire_mem_ratio, ip_ire_cpu_ratio;

1152 static nv_t ire_nv_arr[] = {
1153     { IRE_BROADCAST, "BROADCAST" },
1154     { IRE_LOCAL, "LOCAL" },
1155     { IRE_LOOPBACK, "LOOPBACK" },
1156     { IRE_DEFAULT, "DEFAULT" },
1157     { IRE_PREFIX, "PREFIX" },
1158     { IRE_IF_NORESOLVER, "IF_NORESOL" },
1159     { IRE_IF_RESOLVER, "IF_RESOLV" },
1160     { IRE_IF_CLONE, "IF_CLONE" },
1161     { IRE_HOST, "HOST" },
1162     { IRE_MULTICAST, "MULTICAST" },
1163     { IRE_NOROUTE, "NOROUTE" },
1164     { 0 }
1165 };

1167 nv_t *ire_nv_tbl = ire_nv_arr;

1169 /* Simple ICMP IP Header Template */
1170 static ipha_t icmp_ipha = {
1171     IP_SIMPLE_HDR_VERSION, 0, 0, 0, 0, IPPROTO_ICMP
1172 };

1174 struct module_info ip_mod_info = {
1175     IP_MOD_ID, IP_MOD_NAME, IP_MOD_MINPSZ, IP_MOD_MAXPSZ, IP_MOD_HIWAT,
1176     IP_MOD_LOWAT
1177 };

1179 /*
1180  * Duplicate static symbols within a module confuses mdb; so we avoid the
1181  * problem by making the symbols here distinct from those in udp.c.
1182  */

```

```

1184 /*
1185  * Entry points for IP as a device and as a module.
1186  * We have separate open functions for the /dev/ip and /dev/ip6 devices.
1187  */
1188 static struct qinit iprinitv4 = {
1189     (pfi_t)ip_rput, NULL, ip_openv4, ip_close, NULL,
1190     &ip_mod_info
1191 };
1193 struct qinit iprinitv6 = {
1194     (pfi_t)ip_rput_v6, NULL, ip_openv6, ip_close, NULL,
1195     &ip_mod_info
1196 };
1198 static struct qinit ipwinit = {
1199     (pfi_t)ip_wput_nondata, (pfi_t)ip_wsrv, NULL, NULL, NULL,
1200     &ip_mod_info
1201 };
1203 static struct qinit iplrinit = {
1204     (pfi_t)ip_lrput, NULL, ip_openv4, ip_close, NULL,
1205     &ip_mod_info
1206 };
1208 static struct qinit iplwinit = {
1209     (pfi_t)ip_lwput, NULL, NULL, NULL, NULL,
1210     &ip_mod_info
1211 };
1213 /* For AF_INET aka /dev/ip */
1214 struct streamtab ipinfov4 = {
1215     &iprinitv4, &ipwinit, &iplrinit, &iplwinit
1216 };
1218 /* For AF_INET6 aka /dev/ip6 */
1219 struct streamtab ipinfov6 = {
1220     &iprinitv6, &ipwinit, &iplrinit, &iplwinit
1221 };
1223 #ifdef DEBUG
1224 boolean_t skip_sctp_cksum = B_FALSE;
1225 #endif
1227 /*
1228  * Generate an ICMP fragmentation needed message.
1229  * When called from ip_output side a minimal ip_recv_attr_t needs to be
1230  * constructed by the caller.
1231  */
1232 void
1233 icmp_frag_needed(mblk_t *mp, int mtu, ip_recv_attr_t *ira)
1234 {
1235     icmp_h_t icmp_h;
1236     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
1238     mp = icmp_pkt_err_ok(mp, ira);
1239     if (mp == NULL)
1240         return;
1242     bzero(&icmp_h, sizeof(icmp_h_t));
1243     icmp_h.icmph_type = ICMP_DEST_UNREACHABLE;
1244     icmp_h.icmph_code = ICMP_FRAGMENTATION_NEEDED;
1245     icmp_h.icmph_du_mtu = htons((uint16_t)mtu);
1246     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutFragNeeded);
1247     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);
1249     icmp_pkt(mp, &icmp_h, sizeof(icmp_h_t), ira);

```

```

1250 }
1252 /*
1253  * icmp_inbound_v4 deals with ICMP messages that are handled by IP.
1254  * If the ICMP message is consumed by IP, i.e., it should not be delivered
1255  * to any IPPROTO_ICMP raw sockets, then it returns NULL.
1256  * Likewise, if the ICMP error is misformed (too short, etc), then it
1257  * returns NULL. The caller uses this to determine whether or not to send
1258  * to raw sockets.
1259  *
1260  * All error messages are passed to the matching transport stream.
1261  *
1262  * The following cases are handled by icmp_inbound:
1263  * 1) It needs to send a reply back and possibly delivering it
1264  *    to the "interested" upper clients.
1265  * 2) Return the mblk so that the caller can pass it to the RAW socket clients.
1266  * 3) It needs to change some values in IP only.
1267  * 4) It needs to change some values in IP and upper layers e.g TCP
1268  *    by delivering an error to the upper layers.
1269  *
1270  * We handle the above three cases in the context of IPsec in the
1271  * following way :
1272  *
1273  * 1) Send the reply back in the same way as the request came in.
1274  *    If it came in encrypted, it goes out encrypted. If it came in
1275  *    clear, it goes out in clear. Thus, this will prevent chosen
1276  *    plain text attack.
1277  * 2) The client may or may not expect things to come in secure.
1278  *    If it comes in secure, the policy constraints are checked
1279  *    before delivering it to the upper layers. If it comes in
1280  *    clear, ipsec_inbound_accept_clear will decide whether to
1281  *    accept this in clear or not. In both the cases, if the returned
1282  *    message (IP header + 8 bytes) that caused the icmp message has
1283  *    AH/ESP headers, it is sent up to AH/ESP for validation before
1284  *    sending up. If there are only 8 bytes of returned message, then
1285  *    upper client will not be notified.
1286  * 3) Check with global policy to see whether it matches the constraints.
1287  *    But this will be done only if icmp_accept_messages_in_clear is
1288  *    zero.
1289  * 4) If we need to change both in IP and ULP, then the decision taken
1290  *    while affecting the values in IP and while delivering up to TCP
1291  *    should be the same.
1292  *
1293  *     There are two cases.
1294  *
1295  *     a) If we reject data at the IP layer (ipsec_check_global_policy()
1296  *        failed), we will not deliver it to the ULP, even though they
1297  *        are *willing* to accept in *clear*. This is fine as our global
1298  *        disposition to icmp messages asks us reject the datagram.
1299  *
1300  *     b) If we accept data at the IP layer (ipsec_check_global_policy()
1301  *        succeeded or icmp_accept_messages_in_clear is 1), and not able
1302  *        to deliver it to ULP (policy failed), it can lead to
1303  *        consistency problems. The cases known at this time are
1304  *        ICMP_DESTINATION_UNREACHABLE messages with following code
1305  *        values :
1306  *
1307  *     - ICMP_FRAGMENTATION_NEEDED : IP adapts to the new value
1308  *       and Upper layer rejects. Then the communication will
1309  *       come to a stop. This is solved by making similar decisions
1310  *       at both levels. Currently, when we are unable to deliver
1311  *       to the Upper Layer (due to policy failures) while IP has
1312  *       adjusted dce_pmtu, the next outbound datagram would
1313  *       generate a local ICMP_FRAGMENTATION_NEEDED message - which
1314  *       will be with the right level of protection. Thus the right
1315  *       value will be communicated even if we are not able to

```

```

1316 *      communicate when we get from the wire initially. But this
1317 *      assumes there would be at least one outbound datagram after
1318 *      IP has adjusted its dce_pmtu value. To make things
1319 *      simpler, we accept in clear after the validation of
1320 *      AH/ESP headers.
1321 *
1322 *      - Other ICMP ERRORS : We may not be able to deliver it to the
1323 *      upper layer depending on the level of protection the upper
1324 *      layer expects and the disposition in ipsec_inbound_accept_clear().
1325 *      ipsec_inbound_accept_clear() decides whether a given ICMP error
1326 *      should be accepted in clear when the Upper layer expects secure.
1327 *      Thus the communication may get aborted by some bad ICMP
1328 *      packets.
1329 */
1330 mblk_t *
1331 icmp_inbound_v4(mblk_t *mp, ip_rcv_attr_t *ira)
1332 {
1333     icmp_h_t      *icmph;
1334     ipha_t         *ipha;          /* Outer header */
1335     int            ip_hdr_length; /* Outer header length */
1336     boolean_t      interested;
1337     ipif_t         *ipif;
1338     uint32_t       ts;
1339     uint32_t       tsp;
1340     timestruc_t    now;
1341     ill_t          *ill = ira->ira_ill;
1342     ip_stack_t     *ipst = ill->ill_ipst;
1343     zoneid_t       zoneid = ira->ira_zoneid;
1344     int            len_needed;
1345     mblk_t         *mp_ret = NULL;
1346
1347     ipha = (ipha_t *)mp->b_rptr;
1348
1349     BUMP_MIB(&ipst->ips_icmp_mib, icmpInMsgs);
1350
1351     ip_hdr_length = ira->ira_ip_hdr_length;
1352     if ((mp->b_wptr - mp->b_rptr) < (ip_hdr_length + ICMPH_SIZE)) {
1353         if (ira->ira_pktlen < (ip_hdr_length + ICMPH_SIZE)) {
1354             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
1355             ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
1356             freemsg(mp);
1357             return (NULL);
1358         }
1359         /* Last chance to get real. */
1360         ipha = ip_pullup(mp, ip_hdr_length + ICMPH_SIZE, ira);
1361         if (ipha == NULL) {
1362             BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
1363             freemsg(mp);
1364             return (NULL);
1365         }
1366     }
1367
1368     /* The IP header will always be a multiple of four bytes */
1369     icmph = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1370     ip2dbg(("icmp_inbound_v4: type %d code %d\n", icmph->icmp_type,
1371         icmph->icmp_code));
1372
1373     /*
1374      * We will set "interested" to "true" if we should pass a copy to
1375      * the transport or if we handle the packet locally.
1376      */
1377     interested = B_FALSE;
1378     switch (icmph->icmp_type) {
1379     case ICMP_ECHO_REPLY:
1380         BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchoReps);
1381         break;

```

```

1382     case ICMP_DEST_UNREACHABLE:
1383         if (icmph->icmp_code == ICMP_FRAGMENTATION_NEEDED)
1384             BUMP_MIB(&ipst->ips_icmp_mib, icmpInFragNeeded);
1385         interested = B_TRUE; /* Pass up to transport */
1386         BUMP_MIB(&ipst->ips_icmp_mib, icmpInDestUnreachs);
1387         break;
1388     case ICMP_SOURCE_QUENCH:
1389         interested = B_TRUE; /* Pass up to transport */
1390         BUMP_MIB(&ipst->ips_icmp_mib, icmpInSrcQuenchs);
1391         break;
1392     case ICMP_REDIRECT:
1393         if (!ipst->ips_ip_ignore_redirect)
1394             interested = B_TRUE;
1395         BUMP_MIB(&ipst->ips_icmp_mib, icmpInRedirects);
1396         break;
1397     case ICMP_ECHO_REQUEST:
1398         /*
1399          * Whether to respond to echo requests that come in as IP
1400          * broadcasts or as IP multicast is subject to debate
1401          * (what isn't?). We aim to please, you pick it.
1402          * Default is do it.
1403          */
1404         if (ira->ira_flags & IRAF_MULTICAST) {
1405             /* multicast: respond based on tunable */
1406             interested = ipst->ips_ip_g_resp_to_echo_mcast;
1407         } else if (ira->ira_flags & IRAF_BROADCAST) {
1408             /* broadcast: respond based on tunable */
1409             interested = ipst->ips_ip_g_resp_to_echo_bcst;
1410         } else {
1411             /* unicast: always respond */
1412             interested = B_TRUE;
1413         }
1414         BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchos);
1415         if (!interested) {
1416             /* We never pass these to RAW sockets */
1417             freemsg(mp);
1418             return (NULL);
1419         }
1420
1421         /* Check db_ref to make sure we can modify the packet. */
1422         if (mp->b_datap->db_ref > 1) {
1423             mblk_t *mpl;
1424
1425             mpl = copymsg(mp);
1426             freemsg(mp);
1427             if (!mpl) {
1428                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1429                 return (NULL);
1430             }
1431             mp = mpl;
1432             ipha = (ipha_t *)mp->b_rptr;
1433             icmph = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1434         }
1435         icmph->icmp_type = ICMP_ECHO_REPLY;
1436         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutEchoReps);
1437         icmp_send_reply_v4(mp, ipha, icmph, ira);
1438         return (NULL);
1439
1440     case ICMP_ROUTER_ADVERTISEMENT:
1441     case ICMP_ROUTER_SOLICITATION:
1442         break;
1443     case ICMP_TIME_EXCEEDED:
1444         interested = B_TRUE; /* Pass up to transport */
1445         BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimeExcds);
1446         break;
1447     case ICMP_PARAM_PROBLEM:

```

```

1448         interested = B_TRUE; /* Pass up to transport */
1449         BUMP_MIB(&ipst->ips_icmp_mib, icmpInParmProbs);
1450         break;
1451     case ICMP_TIME_STAMP_REQUEST:
1452         /* Response to Time Stamp Requests is local policy. */
1453         if (ipst->ips_ip_g_resp_to_timestamp) {
1454             if (ira->ira_flags & IRAF_MULTIBROADCAST)
1455                 interested =
1456                     ipst->ips_ip_g_resp_to_timestamp_bcast;
1457             else
1458                 interested = B_TRUE;
1459         }
1460         if (!interested) {
1461             /* We never pass these to RAW sockets */
1462             freemsg(mp);
1463             return (NULL);
1464         }
1465
1466         /* Make sure we have enough of the packet */
1467         len_needed = ip_hdr_length + ICMPH_SIZE +
1468             3 * sizeof(uint32_t);
1469
1470         if (mp->b_wptr - mp->b_rptr < len_needed) {
1471             ipha = ip_pullup(mp, len_needed, ira);
1472             if (ipha == NULL) {
1473                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1474                 ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1475                     mp, ill);
1476                 freemsg(mp);
1477                 return (NULL);
1478             }
1479             /* Refresh following the pullup. */
1480             icmp_h = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1481         }
1482         BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestamps);
1483         /* Check db_ref to make sure we can modify the packet. */
1484         if (mp->b_datap->db_ref > 1) {
1485             mblk_t *mpl;
1486
1487             mpl = copymsg(mp);
1488             freemsg(mp);
1489             if (!mpl) {
1490                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1491                 return (NULL);
1492             }
1493             mp = mpl;
1494             ipha = (ipha_t *)&mp->b_rptr;
1495             icmp_h = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1496         }
1497         icmp_h->icmp_h_type = ICMP_TIME_STAMP_REPLY;
1498         tsp = (uint32_t *)&icmp_h[1];
1499         tsp++; /* Skip past 'originate time' */
1500         /* Compute # of milliseconds since midnight */
1501         gethrestime(&now);
1502         ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
1503             now.tv_nsec / (NANOSEC / MILLISEC);
1504         *tsp++ = htonl(ts); /* Lay in 'receive time' */
1505         *tsp++ = htonl(ts); /* Lay in 'send time' */
1506         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimestampReps);
1507         icmp_send_reply_v4(mp, ipha, icmp_h, ira);
1508         return (NULL);
1509
1510     case ICMP_TIME_STAMP_REPLY:
1511         BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestampReps);
1512         break;
1513     case ICMP_INFO_REQUEST:

```

```

1514         /* Per RFC 1122 3.2.2.7, ignore this. */
1515         case ICMP_INFO_REPLY:
1516             break;
1517         case ICMP_ADDRESS_MASK_REQUEST:
1518             if (ira->ira_flags & IRAF_MULTIBROADCAST) {
1519                 interested =
1520                     ipst->ips_ip_respond_to_address_mask_broadcast;
1521             } else {
1522                 interested = B_TRUE;
1523             }
1524             if (!interested) {
1525                 /* We never pass these to RAW sockets */
1526                 freemsg(mp);
1527                 return (NULL);
1528             }
1529             len_needed = ip_hdr_length + ICMPH_SIZE + IP_ADDR_LEN;
1530             if (mp->b_wptr - mp->b_rptr < len_needed) {
1531                 ipha = ip_pullup(mp, len_needed, ira);
1532                 if (ipha == NULL) {
1533                     BUMP_MIB(ill->ill_ip_mib,
1534                         ipIfStatsInTruncatedPkts);
1535                     ip_drop_input("ipIfStatsInTruncatedPkts", mp,
1536                         ill);
1537                     freemsg(mp);
1538                     return (NULL);
1539                 }
1540                 /* Refresh following the pullup. */
1541                 icmp_h = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1542             }
1543             BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMasks);
1544             /* Check db_ref to make sure we can modify the packet. */
1545             if (mp->b_datap->db_ref > 1) {
1546                 mblk_t *mpl;
1547
1548                 mpl = copymsg(mp);
1549                 freemsg(mp);
1550                 if (!mpl) {
1551                     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1552                     return (NULL);
1553                 }
1554                 mp = mpl;
1555                 ipha = (ipha_t *)&mp->b_rptr;
1556                 icmp_h = (icmp_h_t *)&mp->b_rptr[ip_hdr_length];
1557             }
1558             /*
1559              * Need the ipif with the mask be the same as the source
1560              * address of the mask reply. For unicast we have a specific
1561              * ipif. For multicast/broadcast we only handle onlink
1562              * senders, and use the source address to pick an ipif.
1563              */
1564             ipif = ipif_lookup_addr(ipha->ipha_dst, ill, zoneid, ipst);
1565             if (ipif == NULL) {
1566                 /* Broadcast or multicast */
1567                 ipif = ipif_lookup_remote(ill, ipha->ipha_src, zoneid);
1568                 if (ipif == NULL) {
1569                     freemsg(mp);
1570                     return (NULL);
1571                 }
1572             }
1573             icmp_h->icmp_h_type = ICMP_ADDRESS_MASK_REPLY;
1574             bcopy(&ipif->ipif_net_mask, &icmp_h[1], IP_ADDR_LEN);
1575             ipif_refrele(ipif);
1576             BUMP_MIB(&ipst->ips_icmp_mib, icmpOutAddrMaskReps);
1577             icmp_send_reply_v4(mp, ipha, icmp_h, ira);
1578             return (NULL);

```

```

1580     case ICMP_ADDRESS_MASK_REPLY:
1581         BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMaskReps);
1582         break;
1583     default:
1584         interested = B_TRUE;    /* Pass up to transport */
1585         BUMP_MIB(&ipst->ips_icmp_mib, icmpInUnknowns);
1586         break;
1587 }
1588 /*
1589  * See if there is an ICMP client to avoid an extra copymsg/freemsg
1590  * if there isn't one.
1591  */
1592 if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_ICMP].connf_head != NULL) {
1593     /* If there is an ICMP client and we want one too, copy it. */

1595     if (!interested) {
1596         /* Caller will deliver to RAW sockets */
1597         return (mp);
1598     }
1599     mp_ret = copymsg(mp);
1600     if (mp_ret == NULL) {
1601         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1602         ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1603     }
1604 } else if (!interested) {
1605     /* Neither we nor raw sockets are interested. Drop packet now */
1606     freemsg(mp);
1607     return (NULL);
1608 }

1610 /*
1611  * ICMP error or redirect packet. Make sure we have enough of
1612  * the header and that db_ref == 1 since we might end up modifying
1613  * the packet.
1614  */
1615 if (mp->b_cont != NULL) {
1616     if (ip_pullup(mp, -1, ira) == NULL) {
1617         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1618         ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1619             mp, ill);
1620         freemsg(mp);
1621         return (mp_ret);
1622     }
1623 }

1625 if (mp->b_datap->db_ref > 1) {
1626     mblk_t *mpl;

1628     mpl = copymsg(mp);
1629     if (mpl == NULL) {
1630         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1631         ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1632         freemsg(mp);
1633         return (mp_ret);
1634     }
1635     freemsg(mp);
1636     mp = mpl;
1637 }

1639 /*
1640  * In case mp has changed, verify the message before any further
1641  * processes.
1642  */
1643 ipha = (ipha_t *)mp->b_rptr;
1644 icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1645 if (!icmp_inbound_verify_v4(mp, icmph, ira)) {

```

```

1646         freemsg(mp);
1647         return (mp_ret);
1648     }

1650     switch (icmph->icmph_type) {
1651     case ICMP_REDIRECT:
1652         icmp_redirect_v4(mp, ipha, icmph, ira);
1653         break;
1654     case ICMP_DEST_UNREACHABLE:
1655         if (icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED) {
1656             /* Update DCE and adjust MTU is icmp header if needed */
1657             icmp_inbound_too_big_v4(icmph, ira);
1658         }
1659         /* FALLTHRU */
1660     default:
1661         icmp_inbound_error_fanout_v4(mp, icmph, ira);
1662         break;
1663     }
1664     return (mp_ret);
1665 }

1667 /*
1668  * Send an ICMP echo, timestamp or address mask reply.
1669  * The caller has already updated the payload part of the packet.
1670  * We handle the ICMP checksum, IP source address selection and feed
1671  * the packet into ip_output_simple.
1672  */
1673 static void
1674 icmp_send_reply_v4(mblk_t *mp, ipha_t *ipha, icmph_t *icmph,
1675     ip_rcv_attr_t *ira)
1676 {
1677     uint_t      ip_hdr_length = ira->ira_ip_hdr_length;
1678     ill_t       *ill = ira->ira_ill;
1679     ip_stack_t  *ipst = ill->ill_ipst;
1680     ip_xmit_attr_t ixas;

1682     /* Send out an ICMP packet */
1683     icmph->icmph_checksum = 0;
1684     icmph->icmph_checksum = IP_CSUM(mp, ip_hdr_length, 0);
1685     /* Reset time to live. */
1686     ipha->ipha_ttl = ipst->ips_ip_def_ttl;
1687     {
1688         /* Swap source and destination addresses */
1689         ipaddr_t tmp;

1691         tmp = ipha->ipha_src;
1692         ipha->ipha_src = ipha->ipha_dst;
1693         ipha->ipha_dst = tmp;
1694     }
1695     ipha->ipha_ident = 0;
1696     if (!IS_SIMPLE_IPH(ipha))
1697         icmp_options_update(ipha);

1699     bzero(&ixas, sizeof(ixas));
1700     ixas.ixas_flags = IXAF_BASIC_SIMPLE_V4;
1701     ixas.ixas_zoneid = ira->ira_zoneid;
1702     ixas.ixas_cred = kcred;
1703     ixas.ixas_cpid = NOPID;
1704     ixas.ixas_tsl = ira->ira_tsl;    /* Behave as a multi-level responder */
1705     ixas.ixas_ifindex = 0;
1706     ixas.ixas_ipst = ipst;
1707     ixas.ixas_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;

1709     if (!(ira->ira_flags & IRAF_IPSEC_SECURE)) {
1710         /*
1711          * This packet should go out the same way as it

```

```

1712         * came in i.e in clear, independent of the IPsec policy
1713         * for transmitting packets.
1714         */
1715         ixas.ixaf_flags |= IXAF_NO_IPSEC;
1716     } else {
1717         if (!ipsec_in_to_out(ira, &ixas, mp, ipha, NULL)) {
1718             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1719             /* Note: mp already consumed and ip_drop_packet done */
1720             return;
1721         }
1722     }
1723     if (ira->ira_flags & IRAF_MULTIBROADCAST) {
1724         /*
1725          * Not one of our addresses (IRE_LOCALS), thus we let
1726          * ip_output_simple pick the source.
1727          */
1728         ipha->ipha_src = INADDR_ANY;
1729         ixas.ixaf_flags |= IXAF_SET_SOURCE;
1730     }
1731     /* Should we send with DF and use dce_pmtu? */
1732     if (ipst->ips_ipv4_icmp_return_pmtu) {
1733         ixas.ixaf_flags |= IXAF_PMTU_DISCOVERY;
1734         ipha->ipha_fragment_offset_and_flags |= IPH_DF_HTONS;
1735     }
1737     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutMsgs);
1739     (void) ip_output_simple(mp, &ixas);
1740     ixa_cleanup(&ixas);
1741 }

1743 /*
1744  * Verify the ICMP messages for either for ICMP error or redirect packet.
1745  * The caller should have fully pulled up the message. If it's a redirect
1746  * packet, only basic checks on IP header will be done; otherwise, verify
1747  * the packet by looking at the included ULP header.
1748  *
1749  * Called before icmp_inbound_error_fanout_v4 is called.
1750  */
1751 static boolean_t
1752 icmp_inbound_verify_v4(mblk_t *mp, icmp_t *icmp, ip_rcv_attr_t *ira)
1753 {
1754     ill_t      *ill = ira->ira_ill;
1755     int         hdr_length;
1756     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
1757     conn_t      *connp;
1758     ipha_t      *ipha; /* Inner IP header */

1760     ipha = (ipha_t *)&icmp[1];
1761     if ((uchar_t *)ipha + IP_SIMPLE_HDR_LENGTH > mp->b_wptr)
1762         goto truncated;

1764     hdr_length = IPH_HDR_LENGTH(ipha);

1766     if ((IPH_HDR_VERSION(ipha) != IPV4_VERSION))
1767         goto discard_pkt;

1769     if (hdr_length < sizeof (ipha_t))
1770         goto truncated;

1772     if ((uchar_t *)ipha + hdr_length > mp->b_wptr)
1773         goto truncated;

1775     /*
1776      * Stop here for ICMP_REDIRECT.
1777      */

```

```

1778     if (icmp->icmp_type == ICMP_REDIRECT)
1779         return (B_TRUE);

1781     /*
1782      * ICMP errors only.
1783      */
1784     switch (ipha->ipha_protocol) {
1785     case IPPROTO_UDP:
1786         /*
1787          * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1788          * transport header.
1789          */
1790         if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1791             mp->b_wptr)
1792             goto truncated;
1793         break;
1794     case IPPROTO_TCP: {
1795         tcpha_t      *tcpha;

1797         /*
1798          * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1799          * transport header.
1800          */
1801         if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1802             mp->b_wptr)
1803             goto truncated;

1805         tcpha = (tcpha_t *)((uchar_t *)ipha + hdr_length);
1806         connp = ipcl_tcp_lookup_reversed_ipv4(ipha, tcpha, TCPS_LISTEN,
1807             ipst);
1808         if (connp == NULL)
1809             goto discard_pkt;

1811         if ((connp->conn_verifyicmp != NULL) &&
1812             !connp->conn_verifyicmp(connp, tcpha, icmp, NULL, ira)) {
1813             CONN_DEC_REF(connp);
1814             goto discard_pkt;
1815         }
1816         CONN_DEC_REF(connp);
1817         break;
1818     }
1819     case IPPROTO_SCTP:
1820         /*
1821          * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1822          * transport header.
1823          */
1824         if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1825             mp->b_wptr)
1826             goto truncated;
1827         break;
1828     case IPPROTO_DCCP: {
1829         dccpha_t      *dccpha;

1831         /*
1832          * Verify we have at least ICMP_MIN_TP_HDR_LEN bytes of
1833          * transport header.
1834          */
1835         if ((uchar_t *)ipha + hdr_length + ICMP_MIN_TP_HDR_LEN >
1836             mp->b_wptr)
1837             goto truncated;

1839         cmn_err(CE_NOTE, "icmp_inbound_verify_v4");

1841         dccpha = (dccpha_t *)((uchar_t *)ipha + hdr_length);
1842         /* XXX:DCCP */
1843     }

```

```

1844     connp = ipcl_dccp_lookup_reversed_ipv4(ipha, dccpha,
1845     DCCPS_LISTEN, ipst);
1846     if (connp == NULL) {
1847         goto discard_pkt;
1848     }
1849
1850     if ((connp->conn_verifyicmp != NULL) &&
1851     !connp->conn_verifyicmp(connp, dccpha, icmph, NULL, ira)) {
1852         CONN_DEC_REF(connp);
1853         goto discard_pkt;
1854     }
1855
1856     CONN_DEC_REF(connp);
1857     /*
1858     break;
1859     */
1860 #endif /* ! codereview */
1861     case IPPROTO_ESP:
1862     case IPPROTO_AH:
1863         break;
1864     case IPPROTO_ENCAP:
1865         if ((uchar_t *)ipha + hdr_length + sizeof (ipha_t) >
1866         mp->b_wptr)
1867             goto truncated;
1868         break;
1869     default:
1870         break;
1871     }
1872
1873     return (B_TRUE);
1874
1875 discard_pkt:
1876     /* Bogus ICMP error. */
1877     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1878     return (B_FALSE);
1879
1880 truncated:
1881     /* We pulled up everything already. Must be truncated */
1882     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
1883     ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
1884     return (B_FALSE);
1885 }
1886
1887 /* Table from RFC 1191 */
1888 static int icmp_frag_size_table[] =
1889 { 32000, 17914, 8166, 4352, 2002, 1496, 1006, 508, 296, 68 };
1890
1891 /*
1892  * Process received ICMP Packet too big.
1893  * Just handles the DCE create/update, including using the above table of
1894  * PMTU guesses. The caller is responsible for validating the packet before
1895  * passing it in and also to fanout the ICMP error to any matching transport
1896  * conns. Assumes the message has been fully pulled up and verified.
1897  *
1898  * Before getting here, the caller has called icmp_inbound_verify_v4()
1899  * that should have verified with ULP to prevent undoing the changes we're
1900  * going to make to DCE. For example, TCP might have verified that the packet
1901  * which generated error is in the send window.
1902  *
1903  * In some cases modified this MTU in the ICMP header packet; the caller
1904  * should pass to the matching ULP after this returns.
1905  */
1906 static void
1907 icmp_inbound_too_big_v4(icmph_t *icmph, ip_recv_attr_t *ira)
1908 {
1909     dce_t         *dce;

```

```

1910     int             old_mtu;
1911     int             mtu, orig_mtu;
1912     ipaddr_t        dst;
1913     boolean_t        disable_pmtud;
1914     ill_t            *ill = ira->ira_ill;
1915     ip_stack_t        *ipst = ill->ill_ipst;
1916     uint_t            hdr_length;
1917     ipha_t            *ipha;
1918
1919     /* Caller already pulled up everything. */
1920     ipha = (ipha_t *) &icmph[1];
1921     ASSERT(icmph->icmph_type == ICMP_DEST_UNREACHABLE &&
1922     icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED);
1923     ASSERT(ill != NULL);
1924
1925     hdr_length = IPH_HDR_LENGTH(ipha);
1926
1927     /*
1928     * We handle path MTU for source routed packets since the DCE
1929     * is looked up using the final destination.
1930     */
1931     dst = ip_get_dst(ipha);
1932
1933     dce = dce_lookup_and_add_v4(dst, ipst);
1934     if (dce == NULL) {
1935         /* Couldn't add a unique one - ENOMEM */
1936         ipldbg(("icmp_inbound_too_big_v4: no dce for 0x%x\n",
1937         ntohl(dst)));
1938         return;
1939     }
1940
1941     /* Check for MTU discovery advice as described in RFC 1191 */
1942     mtu = ntohs(icmph->icmph_du_mtu);
1943     orig_mtu = mtu;
1944     disable_pmtud = B_FALSE;
1945
1946     mutex_enter(&dce->dce_lock);
1947     if (dce->dce_flags & DCEF_PMTU)
1948         old_mtu = dce->dce_pmtu;
1949     else
1950         old_mtu = ill->ill_mtu;
1951
1952     if (icmph->icmph_du_zero != 0 || mtu < ipst->ips_ip_pmtu_min) {
1953         uint32_t length;
1954         int         i;
1955
1956         /*
1957         * Use the table from RFC 1191 to figure out
1958         * the next "plateau" based on the length in
1959         * the original IP packet.
1960         */
1961         length = ntohs(ipha->ipha_length);
1962         DTRACE_PROBE2(ip4_pmtu_guess, dce_t *, dce,
1963         uint32_t, length);
1964         if (old_mtu <= length &&
1965         old_mtu >= length - hdr_length) {
1966             /*
1967             * Handle broken BSD 4.2 systems that
1968             * return the wrong ipha_length in ICMP
1969             * errors.
1970             */
1971             ipldbg(("Wrong mtu: sent %d, dce %d\n",
1972             length, old_mtu));
1973             length -= hdr_length;
1974         }
1975         for (i = 0; i < A_CNT(icmp_frag_size_table); i++) {

```



```

1976         if (length > icmp_frag_size_table[i])
1977             break;
1978     }
1979     if (i == A_CNT(icmp_frag_size_table)) {
1980         /* Smaller than IP_MIN_MTU! */
1981         ipldbg(("Too big for packet size %d\n",
1982             length));
1983         disable_pmtud = B_TRUE;
1984         mtu = ipst->ips_ip_pmtu_min;
1985     } else {
1986         mtu = icmp_frag_size_table[i];
1987         ipldbg(("Calculated mtu %d, packet size %d, "
1988             "before %d\n", mtu, length, old_mtu));
1989         if (mtu < ipst->ips_ip_pmtu_min) {
1990             mtu = ipst->ips_ip_pmtu_min;
1991             disable_pmtud = B_TRUE;
1992         }
1993     }
1994 }
1995 if (disable_pmtud)
1996     dce->dce_flags |= DCEF_TOO_SMALL_PMTU;
1997 else
1998     dce->dce_flags &= ~DCEF_TOO_SMALL_PMTU;
1999
2000 dce->dce_pmtu = MIN(old_mtu, mtu);
2001 /* Prepare to send the new max frag size for the ULP. */
2002 icmp_h->icmp_hu_zero = 0;
2003 icmp_h->icmp_hu_mtu = htons((uint16_t)dce->dce_pmtu);
2004 DTRACE_PROBE4(ip4_pmtu_change, icmp_h_t *, icmp_h, dce_t *,
2005     dce, int, orig_mtu, int, mtu);
2006
2007 /* We now have a PMTU for sure */
2008 dce->dce_flags |= DCEF_PMTU;
2009 dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
2010 mutex_exit(&dce->dce_lock);
2011 /*
2012  * After dropping the lock the new value is visible to everyone.
2013  * Then we bump the generation number so any cached values reinspect
2014  * the dce_t.
2015  */
2016 dce_increment_generation(dce);
2017 dce_refrele(dce);
2018 }
2019
2020 /*
2021  * If the packet in error is Self-Encapsulated, icmp_inbound_error_fanout_v4
2022  * calls this function.
2023  */
2024 static mblk_t *
2025 icmp_inbound_self_encap_error_v4(mblk_t *mp, ipha_t *ipha, ipha_t *in_ipha)
2026 {
2027     int length;
2028
2029     ASSERT(mp->b_datap->db_type == M_DATA);
2030
2031     /* icmp_inbound_v4 has already pulled up the whole error packet */
2032     ASSERT(mp->b_cont == NULL);
2033
2034     /*
2035      * The length that we want to overlay is the inner header
2036      * and what follows it.
2037      */
2038     length = msgdsize(mp) - ((uchar_t *)in_ipha - mp->b_rptr);
2039
2040     /*
2041      * Overlay the inner header and whatever follows it over the

```

```

2042     * outer header.
2043     */
2044     bcopy((uchar_t *)in_ipha, (uchar_t *)ipha, length);
2045
2046     /* Adjust for what we removed */
2047     mp->b_wptr -= (uchar_t *)in_ipha - (uchar_t *)ipha;
2048     return (mp);
2049 }
2050
2051 /*
2052  * Try to pass the ICMP message upstream in case the ULP cares.
2053  *
2054  * If the packet that caused the ICMP error is secure, we send
2055  * it to AH/ESP to make sure that the attached packet has a
2056  * valid association. ipha in the code below points to the
2057  * IP header of the packet that caused the error.
2058  *
2059  * For IPsec cases, we let the next-layer-up (which has access to
2060  * cached policy on the conn_t, or can query the SPD directly)
2061  * subtract out any IPsec overhead if they must. We therefore make no
2062  * adjustments here for IPsec overhead.
2063  *
2064  * IFN could have been generated locally or by some router.
2065  *
2066  * LOCAL : ire_send_wire (before calling ipsec_out_process) can call
2067  * icmp_frag_needed/icmp_pkt2big_v6 to generated a local IFN.
2068  * This happens because IP adjusted its value of MTU on an
2069  * earlier IFN message and could not tell the upper layer,
2070  * the new adjusted value of MTU e.g. Packet was encrypted
2071  * or there was not enough information to fanout to upper
2072  * layers. Thus on the next outbound datagram, ire_send_wire
2073  * generates the IFN, where IPsec processing has *not* been
2074  * done.
2075  *
2076  * Note that we retain ix_a_fragsize across IPsec thus once
2077  * we have picking ix_a_fragsize and entered ipsec_out_process we do
2078  * no change the fragsize even if the path MTU changes before
2079  * we reach ip_output_post_ipsec.
2080  *
2081  * In the local case, IRAF_LOOPBACK will be set indicating
2082  * that IFN was generated locally.
2083  *
2084  * ROUTER : IFN could be secure or non-secure.
2085  *
2086  * SECURE : We use the IPSEC_IN to fanout to AH/ESP if the
2087  * packet in error has AH/ESP headers to validate the AH/ESP
2088  * headers. AH/ESP will verify whether there is a valid SA or
2089  * not and send it back. We will fanout again if we have more
2090  * data in the packet.
2091  *
2092  * If the packet in error does not have AH/ESP, we handle it
2093  * like any other case.
2094  *
2095  * NON_SECURE : If the packet in error has AH/ESP headers, we send it
2096  * up to AH/ESP for validation. AH/ESP will verify whether there is a
2097  * valid SA or not and send it back. We will fanout again if
2098  * we have more data in the packet.
2099  *
2100  * If the packet in error does not have AH/ESP, we handle it
2101  * like any other case.
2102  *
2103  * The caller must have called icmp_inbound_verify_v4.
2104  */
2105 static void
2106 icmp_inbound_error_fanout_v4(mblk_t *mp, icmp_h_t *icmp_h, ip_rcv_attr_t *ira)
2107 {

```

```

2108     uint16_t      *up;      /* Pointer to ports in ULP header */
2109     uint32_t      ports;    /* reversed ports for fanout */
2110     ipha_t        riph;     /* With reversed addresses */
2111     ipha_t        *ipha;    /* Inner IP header */
2112     uint_t        hdr_length; /* Inner IP header length */
2113     tcpa_t        *tcpa;
2114     connp_t       *connp;
2115     ill_t         *ill = ira->ira_ill;
2116     ip_stack_t    *ipst = ill->ill_ipst;
2117     ipsec_stack_t *ipss = ipst->ipsec_stack->netstack_ipsec;
2118     ill_t         *rill = ira->ira_rill;

2120     /* Caller already pulled up everything. */
2121     ipha = (ipha_t *) &icmph[1];
2122     ASSERT((uchar_t *) &ipha[1] <= mp->b_wptr);
2123     ASSERT(mp->b_cont == NULL);

2125     hdr_length = IPH_HDR_LENGTH(ipha);
2126     ira->ira_protocol = ipha->ipha_protocol;

2128     /*
2129      * We need a separate IP header with the source and destination
2130      * addresses reversed to do fanout/classification because the ipha in
2131      * the ICMP error is in the form we sent it out.
2132      */
2133     riph.ipha_src = ipha->ipha_dst;
2134     riph.ipha_dst = ipha->ipha_src;
2135     riph.ipha_protocol = ipha->ipha_protocol;
2136     riph.ipha_version_and_hdr_length = ipha->ipha_version_and_hdr_length;

2138     ip2dbg(("icmp_inbound_error_v4: proto %d %x to %x: %d/%d\n",
2139           riph.ipha_protocol, ntohs(ipha->ipha_src),
2140           ntohs(ipha->ipha_dst),
2141           icmph->icmph_type, icmph->icmph_code));

2143     switch (ipha->ipha_protocol) {
2144     case IPPROTO_UDP:
2145         up = (uint16_t *) ((uchar_t *) ipha + hdr_length);

2147         /* Attempt to find a client stream based on port. */
2148         ip2dbg(("icmp_inbound_error_v4: UDP ports %d to %d\n",
2149               ntohs(up[0]), ntohs(up[1])));

2151         /* Note that we send error to all matches. */
2152         ira->ira_flags |= IRAF_ICMP_ERROR;
2153         ip_fanout_udp_multi_v4(mp, &riph, up[0], up[1], ira);
2154         ira->ira_flags &= ~IRAF_ICMP_ERROR;
2155         return;

2157     case IPPROTO_TCP:
2158         /*
2159          * Find a TCP client stream for this packet.
2160          * Note that we do a reverse lookup since the header is
2161          * in the form we sent it out.
2162          */
2163         tcpa = (tcpa_t *) ((uchar_t *) ipha + hdr_length);
2164         connp = ipcl_tcp_lookup_reversed_ip4(ipha, tcpa, TCPS_LISTEN,
2165               ipst);
2166         if (connp == NULL)
2167             goto discard_pkt;

2169         if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2170             (ira->ira_flags & IRAF_IPSEC_SECURE)) {
2171             mp = ipsec_check_inbound_policy(mp, connp,
2172                   ipha, NULL, ira);
2173             if (mp == NULL) {

```

```

2174         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2175         /* Note that mp is NULL */
2176         ip_drop_input("ipIfStatsInDiscards", mp, ill);
2177         CONN_DEC_REF(connp);
2178         return;
2179     }
2180 }

2182     ira->ira_flags |= IRAF_ICMP_ERROR;
2183     ira->ira_ill = ira->ira_rill = NULL;
2184     if (IPCL_IS_TCP(connp)) {
2185         SQUEUE_ENTER_ONE(connp->conn_sq, mp,
2186               connp->conn_recvicmp, connp, ira, SQ_FILL,
2187               SQTAG_TCP_INPUT_ICMP_ERR);
2188     } else {
2189         /* Not TCP; must be SOCK_RAW, IPPROTO_TCP */
2190         (connp->conn_recv)(connp, mp, NULL, ira);
2191         CONN_DEC_REF(connp);
2192     }
2193     ira->ira_ill = ill;
2194     ira->ira_rill = rill;
2195     ira->ira_flags &= ~IRAF_ICMP_ERROR;
2196     return;

2198     case IPPROTO_SCTP:
2199         up = (uint16_t *) ((uchar_t *) ipha + hdr_length);
2200         /* Find a SCTP client stream for this packet. */
2201         ((uint16_t *) &ports)[0] = up[1];
2202         ((uint16_t *) &ports)[1] = up[0];

2204         ira->ira_flags |= IRAF_ICMP_ERROR;
2205         ip_fanout_sctp(mp, &riph, NULL, ports, ira);
2206         ira->ira_flags &= ~IRAF_ICMP_ERROR;
2207         return;

2209     case IPPROTO_DCCP:
2210         cmn_err(CE_NOTE, "icmp_inbound_error_fanout_v4");
2211         return;

2213 #endif /* !codereview */
2214     case IPPROTO_ESP:
2215     case IPPROTO_AH:
2216         if (!ipsec_loaded(ipss)) {
2217             ip_proto_not_sup(mp, ira);
2218             return;
2219         }

2221         if (ipha->ipha_protocol == IPPROTO_ESP)
2222             mp = ipsecsp_icmp_error(mp, ira);
2223         else
2224             mp = ipsecah_icmp_error(mp, ira);
2225         if (mp == NULL)
2226             return;

2228         /* Just in case ipsec didn't preserve the NULL b_cont */
2229         if (mp->b_cont != NULL) {
2230             if (!pullupmsg(mp, -1))
2231                 goto discard_pkt;
2232         }

2234         /*
2235          * Note that ira_pktlen and ira_ip_hdr_length are no longer
2236          * correct, but we don't use them any more here.
2237          *
2238          * If succesful, the mp has been modified to not include
2239          * the ESP/AH header so we can fanout to the ULP's icmp

```

```

2240     * error handler.
2241     */
2242     if (mp->b_wptr - mp->b_rptr < IP_SIMPLE_HDR_LENGTH)
2243         goto truncated;

2245     /* Verify the modified message before any further processes. */
2246     ipha = (ipha_t *)mp->b_rptr;
2247     hdr_length = IPH_HDR_LENGTH(ipha);
2248     icmph = (icmph_t *)&mp->b_rptr[hdr_length];
2249     if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
2250         freemsg(mp);
2251         return;
2252     }

2254     icmp_inbound_error_fanout_v4(mp, icmph, ira);
2255     return;

2257 case IPPROTO_ENCAP: {
2258     /* Look for self-encapsulated packets that caused an error */
2259     ipha_t *in_ipha;

2261     /*
2262     * Caller has verified that length has to be
2263     * at least the size of IP header.
2264     */
2265     ASSERT(hdr_length >= sizeof (ipha_t));
2266     /*
2267     * Check the sanity of the inner IP header like
2268     * we did for the outer header.
2269     */
2270     in_ipha = (ipha_t *)((uchar_t *)ipha + hdr_length);
2271     if ((IPH_HDR_VERSION(in_ipha) != IPV4_VERSION)) {
2272         goto discard_pkt;
2273     }
2274     if (IPH_HDR_LENGTH(in_ipha) < sizeof (ipha_t)) {
2275         goto discard_pkt;
2276     }
2277     /* Check for Self-encapsulated tunnels */
2278     if (in_ipha->ipha_src == ipha->ipha_src &&
2279         in_ipha->ipha_dst == ipha->ipha_dst) {

2281         mp = icmp_inbound_self_encap_error_v4(mp, ipha,
2282             in_ipha);
2283         if (mp == NULL)
2284             goto discard_pkt;

2286     /*
2287     * Just in case self_encap didn't preserve the NULL
2288     * b_cont
2289     */
2290     if (mp->b_cont != NULL) {
2291         if (!pullupmsg(mp, -1))
2292             goto discard_pkt;
2293     }
2294     /*
2295     * Note that ira_pktlen and ira_ip_hdr_length are no
2296     * longer correct, but we don't use them any more here.
2297     */
2298     if (mp->b_wptr - mp->b_rptr < IP_SIMPLE_HDR_LENGTH)
2299         goto truncated;

2301     /*
2302     * Verify the modified message before any further
2303     * processes.
2304     */
2305     ipha = (ipha_t *)mp->b_rptr;

```

```

2306         hdr_length = IPH_HDR_LENGTH(ipha);
2307         icmph = (icmph_t *)&mp->b_rptr[hdr_length];
2308         if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
2309             freemsg(mp);
2310             return;
2311         }

2313     /*
2314     * The packet in error is self-encapsulated.
2315     * And we are finding it further encapsulated
2316     * which we could not have possibly generated.
2317     */
2318     if (ipha->ipha_protocol == IPPROTO_ENCAP) {
2319         goto discard_pkt;
2320     }
2321     icmp_inbound_error_fanout_v4(mp, icmph, ira);
2322     return;
2323 }
2324 /* No self-encapsulated */
2325 /* FALLTHRU */
2326 }
2327 case IPPROTO_IPV6:
2328     if ((connp = ipcl_iptun_classify_v4(&ripha.ipha_src,
2329         &ripha.ipha_dst, ipst)) != NULL) {
2330         ira->ira_flags |= IRAF_ICMP_ERROR;
2331         connp->conn_recvcmp(connp, mp, NULL, ira);
2332         CONN_DEC_REF(connp);
2333         ira->ira_flags &= ~IRAF_ICMP_ERROR;
2334         return;
2335     }
2336     /*
2337     * No IP tunnel is interested, fallthrough and see
2338     * if a raw socket will want it.
2339     */
2340     /* FALLTHRU */
2341 default:
2342     ira->ira_flags |= IRAF_ICMP_ERROR;
2343     ip_fanout_proto_v4(mp, &ripha, ira);
2344     ira->ira_flags &= ~IRAF_ICMP_ERROR;
2345     return;
2346 }
2347 /* NOTREACHED */
2348 discard_pkt:
2349     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2350     ipldbg(("icmp_inbound_error_fanout_v4: drop pkt\n"));
2351     ip_drop_input("ipIfStatsInDiscards", mp, ill);
2352     freemsg(mp);
2353     return;

2355 truncated:
2356     /* We pulled up everything already. Must be truncated */
2357     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
2358     ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
2359     freemsg(mp);
2360 }

2362 /*
2363 * Common IP options parser.
2364 *
2365 * Setup routine: fill in *optp with options-parsing state, then
2366 * tail-call ipoptp_next to return the first option.
2367 */
2368 uint8_t
2369 ipoptp_first(ipoptp_t *optp, ipha_t *ipha)
2370 {
2371     uint32_t totallen; /* total length of all options */

```

```

2373     totallen = ipha->ipha_version_and_hdr_length -
2374         (uint8_t)((IP_VERSION < 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
2375     totallen <= 2;
2376     optp->iptptp_next = (uint8_t *)&ipha[1];
2377     optp->iptptp_end = optp->iptptp_next + totallen;
2378     optp->iptptp_flags = 0;
2379     return (iptptp_next(optp));
2380 }

2382 /* Like above but without an ipha_t */
2383 uint8_t
2384 iptptp_first2(iptptp_t *optp, uint32_t totallen, uint8_t *opt)
2385 {
2386     optp->iptptp_next = opt;
2387     optp->iptptp_end = optp->iptptp_next + totallen;
2388     optp->iptptp_flags = 0;
2389     return (iptptp_next(optp));
2390 }

2392 /*
2393  * Common IP options parser: extract next option.
2394  */
2395 uint8_t
2396 iptptp_next(iptptp_t *optp)
2397 {
2398     uint8_t *end = optp->iptptp_end;
2399     uint8_t *cur = optp->iptptp_next;
2400     uint8_t opt, len, pointer;

2402     /*
2403      * If cur > end already, then the iptptp_end or iptptp_next pointer
2404      * has been corrupted.
2405      */
2406     ASSERT(cur <= end);

2408     if (cur == end)
2409         return (ILOPT_EOL);

2411     opt = cur[ILOPT_OPTVAL];

2413     /*
2414      * Skip any NOP options.
2415      */
2416     while (opt == ILOPT_NOP) {
2417         cur++;
2418         if (cur == end)
2419             return (ILOPT_EOL);
2420         opt = cur[ILOPT_OPTVAL];
2421     }

2423     if (opt == ILOPT_EOL)
2424         return (ILOPT_EOL);

2426     /*
2427      * Option requiring a length.
2428      */
2429     if ((cur + 1) >= end) {
2430         optp->iptptp_flags |= IPOPTP_ERROR;
2431         return (ILOPT_EOL);
2432     }
2433     len = cur[ILOPT_OLEN];
2434     if (len < 2) {
2435         optp->iptptp_flags |= IPOPTP_ERROR;
2436         return (ILOPT_EOL);
2437     }

```

```

2438     optp->iptptp_cur = cur;
2439     optp->iptptp_len = len;
2440     optp->iptptp_next = cur + len;
2441     if (cur + len > end) {
2442         optp->iptptp_flags |= IPOPTP_ERROR;
2443         return (ILOPT_EOL);
2444     }

2446     /*
2447      * For the options which require a pointer field, make sure
2448      * its there, and make sure it points to either something
2449      * inside this option, or the end of the option.
2450      */
2451     switch (opt) {
2452     case ILOPT_RR:
2453     case ILOPT_TS:
2454     case ILOPT_LSRR:
2455     case ILOPT_SSRR:
2456         if (len <= ILOPT_OFFSET) {
2457             optp->iptptp_flags |= IPOPTP_ERROR;
2458             return (opt);
2459         }
2460         pointer = cur[ILOPT_OFFSET];
2461         if (pointer - 1 > len) {
2462             optp->iptptp_flags |= IPOPTP_ERROR;
2463             return (opt);
2464         }
2465         break;
2466     }

2468     /*
2469      * Sanity check the pointer field based on the type of the
2470      * option.
2471      */
2472     switch (opt) {
2473     case ILOPT_RR:
2474     case ILOPT_SSRR:
2475     case ILOPT_LSRR:
2476         if (pointer < ILOPT_MINOFF_SR)
2477             optp->iptptp_flags |= IPOPTP_ERROR;
2478         break;
2479     case ILOPT_TS:
2480         if (pointer < ILOPT_MINOFF_IT)
2481             optp->iptptp_flags |= IPOPTP_ERROR;
2482     /*
2483      * Note that the Internet Timestamp option also
2484      * contains two four bit fields (the Overflow field,
2485      * and the Flag field), which follow the pointer
2486      * field. We don't need to check that these fields
2487      * fall within the length of the option because this
2488      * was implicitly done above. We've checked that the
2489      * pointer value is at least ILOPT_MINOFF_IT, and that
2490      * it falls within the option. Since ILOPT_MINOFF_IT >
2491      * ILOPT_POS_OV_FLG, we don't need the explicit check.
2492      */
2493     case ILOPT_POS_OV_FLG:
2494         ASSERT(len > ILOPT_POS_OV_FLG);
2495         break;
2496     }

2497     return (opt);
2498 }

2500 /*
2501  * Use the outgoing IP header to create an IP_OPTIONS option the way
2502  * it was passed down from the application.
2503  */

```

```

2504 * This is compatible with BSD in that it returns
2505 * the reverse source route with the final destination
2506 * as the last entry. The first 4 bytes of the option
2507 * will contain the final destination.
2508 */
2509 int
2510 ip_opt_get_user(conn_t *connp, uchar_t *buf)
2511 {
2512     ipoptp_t      opts;
2513     uchar_t       *opt;
2514     uint8_t       optval;
2515     uint8_t       optlen;
2516     uint32_t      len = 0;
2517     uchar_t       *buf1 = buf;
2518     uint32_t      totallen;
2519     ipaddr_t      dst;
2520     ip_pkt_t      *ipp = &connp->conn_xmit_ipp;

2522     if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
2523         return (0);

2525     totallen = ipp->ipp_ipv4_options_len;
2526     if (totallen & 0x3)
2527         return (0);

2529     buf += IP_ADDR_LEN; /* Leave room for final destination */
2530     len += IP_ADDR_LEN;
2531     bzero(buf1, IP_ADDR_LEN);

2533     dst = connp->conn_faddr_v4;

2535     for (optval = ipoptp_first2(&opts, totallen, ipp->ipp_ipv4_options);
2536          optval != IPOPT_EOL;
2537          optval = ipoptp_next(&opts)) {
2538         int      off;

2540         opt = opts.ipoptp_cur;
2541         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
2542             break;
2543         }
2544         optlen = opts.ipoptp_len;

2546         switch (optval) {
2547             case IPOPT_SSRR:
2548             case IPOPT_LSRR:

2550                 /*
2551                  * Insert destination as the first entry in the source
2552                  * route and move down the entries on step.
2553                  * The last entry gets placed at buf1.
2554                  */
2555                 buf[IPOPT_OPTVAL] = optval;
2556                 buf[IPOPT_OLEN] = optlen;
2557                 buf[IPOPT_OFFSET] = optlen;

2559                 off = optlen - IP_ADDR_LEN;
2560                 if (off < 0) {
2561                     /* No entries in source route */
2562                     break;
2563                 }
2564                 /* Last entry in source route if not already set */
2565                 if (dst == INADDR_ANY)
2566                     bcopy(opt + off, buf1, IP_ADDR_LEN);
2567                 off -= IP_ADDR_LEN;

2569                 while (off > 0) {

```

```

2570             bcopy(opt + off,
2571                   buf + off + IP_ADDR_LEN,
2572                   IP_ADDR_LEN);
2573             off -= IP_ADDR_LEN;
2574         }
2575         /* ipha_dst into first slot */
2576         bcopy(&dst, buf + off + IP_ADDR_LEN,
2577             IP_ADDR_LEN);
2578         buf += optlen;
2579         len += optlen;
2580         break;

2582     default:
2583         bcopy(opt, buf, optlen);
2584         buf += optlen;
2585         len += optlen;
2586         break;
2587     }
2588 }
2589 done:
2590     /* Pad the resulting options */
2591     while (len & 0x3) {
2592         *buf++ = IPOPT_EOL;
2593         len++;
2594     }
2595     return (len);
2596 }

2598 /*
2599  * Update any record route or timestamp options to include this host.
2600  * Reverse any source route option.
2601  * This routine assumes that the options are well formed i.e. that they
2602  * have already been checked.
2603  */
2604 static void
2605 icmp_options_update(ipha_t *ipha)
2606 {
2607     ipoptp_t      opts;
2608     uchar_t       *opt;
2609     uint8_t       optval;
2610     ipaddr_t      src; /* Our local address */
2611     ipaddr_t      dst;

2613     ip2dbg(("icmp_options_update\n"));
2614     src = ipha->ipha_src;
2615     dst = ipha->ipha_dst;

2617     for (optval = ipoptp_first(&opts, ipha);
2618          optval != IPOPT_EOL;
2619          optval = ipoptp_next(&opts)) {
2620         ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
2621         opt = opts.ipoptp_cur;
2622         ip2dbg(("icmp_options_update: opt %d, len %d\n",
2623             optval, opts.ipoptp_len));
2624         switch (optval) {
2625             case IPOPT_SSRR:
2626             case IPOPT_LSRR:
2627                 /*
2628                  * Reverse the source route. The first entry
2629                  * should be the next to last one in the current
2630                  * source route (the last entry is our address).
2631                  * The last entry should be the final destination.
2632                  */
2633                 off1 = IPOPT_MINOFF_SR - 1;
2634                 off2 = opt[IPOPT_OFFSET] - IP_ADDR_LEN - 1;

```

```

2636         if (off2 < 0) {
2637             /* No entries in source route */
2638             ipldbg(("icmp_options_update: bad src route\n"));
2639             break;
2640         }
2641         bcopy((char *)opt + off2, &dst, IP_ADDR_LEN);
2642         bcopy(&ipha->ipha_dst, (char *)opt + off2, IP_ADDR_LEN);
2643         bcopy(&dst, &ipha->ipha_dst, IP_ADDR_LEN);
2644         off2 -= IP_ADDR_LEN;
2645
2646         while (off1 < off2) {
2647             bcopy((char *)opt + off1, &src, IP_ADDR_LEN);
2648             bcopy((char *)opt + off2, (char *)opt + off1,
2649                 IP_ADDR_LEN);
2650             bcopy(&src, (char *)opt + off2, IP_ADDR_LEN);
2651             off1 += IP_ADDR_LEN;
2652             off2 -= IP_ADDR_LEN;
2653         }
2654         opt[IPOPT_OFFSET] = IPOPT_MINOFF_SR;
2655         break;
2656     }
2657 }
2658 }
2659 }

```

```

2661 /*
2662  * Process received ICMP Redirect messages.
2663  * Assumes the caller has verified that the headers are in the pulled up mblk.
2664  * Consumes mp.
2665  */
2666 static void
2667 icmp_redirect_v4(mblk_t *mp, ipha_t *ipha, icmph_t *icmph, ip_rcv_attr_t *ira)
2668 {
2669     ire_t      *ire, *nire;
2670     ire_t      *prev_ire;
2671     ipaddr_t    src, dst, gateway;
2672     ip_stack_t  *ipst = ira->ira_ill->ill_ipst;
2673     ipha_t      *inner_ipha; /* Inner IP header */

```

```

2675     /* Caller already pulled up everything. */
2676     inner_ipha = (ipha_t *)&icmph[1];
2677     src = ipha->ipha_src;
2678     dst = inner_ipha->ipha_dst;
2679     gateway = icmph->icmph_rd_gateway;
2680     /* Make sure the new gateway is reachable somehow. */
2681     ire = ire_fhtable_lookup_v4(gateway, 0, 0, IRE_ONLINK, NULL,
2682         ALL_ZONES, NULL, MATCH_IRE_TYPE, 0, ipst, NULL);
2683     /*
2684      * Make sure we had a route for the dest in question and that
2685      * that route was pointing to the old gateway (the source of the
2686      * redirect packet.)
2687      * We do longest match and then compare ire_gateway_addr below.
2688      */
2689     prev_ire = ire_fhtable_lookup_v4(dst, 0, 0, 0, NULL, ALL_ZONES,
2690         NULL, MATCH_IRE_DSTONLY, 0, ipst, NULL);
2691     /*
2692      * Check that
2693      * the redirect was not from ourselves
2694      * the new gateway and the old gateway are directly reachable
2695      */
2696     if (prev_ire == NULL || ire == NULL ||
2697         (prev_ire->ire_type & (IRE_LOCAL | IRE_LOOPBACK)) ||
2698         (prev_ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE)) ||
2699         !(ire->ire_type & IRE_IF_ALL) ||
2700         prev_ire->ire_gateway_addr != src) {
2701         BUMP_MIB(&ipst->ips_icmp_mib, icmpInBadRedirects);

```

```

2702         ip_drop_input("icmpInBadRedirects - ire", mp, ira->ira_ill);
2703         freemsg(mp);
2704         if (ire != NULL)
2705             ire_refrele(ire);
2706         if (prev_ire != NULL)
2707             ire_refrele(prev_ire);
2708         return;
2709     }

```

```

2711     ire_refrele(prev_ire);
2712     ire_refrele(ire);

```

```

2714     /*
2715      * TODO: more precise handling for cases 0, 2, 3, the latter two
2716      * require TOS routing
2717      */
2718     switch (icmph->icmph_code) {
2719     case 0:
2720     case 1:
2721         /* TODO: TOS specificity for cases 2 and 3 */
2722     case 2:
2723     case 3:
2724         break;
2725     default:
2726         BUMP_MIB(&ipst->ips_icmp_mib, icmpInBadRedirects);
2727         ip_drop_input("icmpInBadRedirects - code", mp, ira->ira_ill);
2728         freemsg(mp);
2729         return;
2730     }
2731     /*
2732      * Create a Route Association. This will allow us to remember that
2733      * someone we believe told us to use the particular gateway.
2734      */
2735     ire = ire_create(
2736         (uchar_t *)&dst, /* dest addr */
2737         (uchar_t *)&ip_g_all_ones, /* mask */
2738         (uchar_t *)&gateway, /* gateway addr */
2739         IRE_HOST,
2740         NULL, /* ill */
2741         ALL_ZONES,
2742         (RTF_DYNAMIC | RTF_GATEWAY | RTF_HOST),
2743         NULL, /* tsol_gc_t */
2744         ipst);

```

```

2746     if (ire == NULL) {
2747         freemsg(mp);
2748         return;
2749     }
2750     nire = ire_add(ire);
2751     /* Check if it was a duplicate entry */
2752     if (nire != NULL && nire != ire) {
2753         ASSERT(nire->ire_identical_ref > 1);
2754         ire_delete(nire);
2755         ire_refrele(nire);
2756         nire = NULL;
2757     }
2758     ire = nire;
2759     if (ire != NULL) {
2760         ire_refrele(ire); /* Held in ire_add */

```

```

2762         /* tell routing sockets that we received a redirect */
2763         ip_rts_change(RTM_REDIRECT, dst, gateway, IP_HOST_MASK, 0, src,
2764             (RTF_DYNAMIC | RTF_GATEWAY | RTF_HOST), 0,
2765             (RTA_DST | RTA_GATEWAY | RTA_NETMASK | RTA_AUTHOR), ipst);
2766     }

```

```

2768 /*
2769  * Delete any existing IRE_HOST type redirect ires for this destination.
2770  * This together with the added IRE has the effect of
2771  * modifying an existing redirect.
2772  */
2773 prev_ire = ire_fhtable_lookup_v4(dst, 0, src, IRE_HOST, NULL,
2774 ALL_ZONES, NULL, (MATCH_IRE_GW | MATCH_IRE_TYPE), 0, ipst, NULL);
2775 if (prev_ire != NULL) {
2776     if (prev_ire->ire_flags & RTF_DYNAMIC)
2777         ire_delete(prev_ire);
2778     ire_refrele(prev_ire);
2779 }
2781 freemsg(mp);
2782 }
2784 /*
2785  * Generate an ICMP parameter problem message.
2786  * When called from ip_output side a minimal ip_rcv_attr_t needs to be
2787  * constructed by the caller.
2788  */
2789 static void
2790 icmp_param_problem(mblk_t *mp, uint8_t ptr, ip_rcv_attr_t *ira)
2791 {
2792     icmp_h_t icmp_h;
2793     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
2795     mp = icmp_pkt_err_ok(mp, ira);
2796     if (mp == NULL)
2797         return;
2799     bzero(&icmp_h, sizeof(icmp_h_t));
2800     icmp_h.icmph_type = ICMP_PARAM_PROBLEM;
2801     icmp_h.icmph_pp_ptr = ptr;
2802     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutParmProbs);
2803     icmp_pkt(mp, &icmp_h, sizeof(icmp_h_t), ira);
2804 }
2806 /*
2807  * Build and ship an IPv4 ICMP message using the packet data in mp, and
2808  * the ICMP header pointed to by "stuff". (May be called as writer.)
2809  * Note: assumes that icmp_pkt_err_ok has been called to verify that
2810  * an icmp error packet can be sent.
2811  * Assigns an appropriate source address to the packet. If ipha_dst is
2812  * one of our addresses use it for source. Otherwise let ip_output_simple
2813  * pick the source address.
2814  */
2815 static void
2816 icmp_pkt(mblk_t *mp, void *stuff, size_t len, ip_rcv_attr_t *ira)
2817 {
2818     ipaddr_t dst;
2819     icmp_h_t *icmph;
2820     ipha_t *ipha;
2821     uint_t len_needed;
2822     size_t msg_len;
2823     mblk_t *mpl;
2824     ipaddr_t src;
2825     ire_t *ire;
2826     ip_xmit_attr_t ixas;
2827     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
2829     ipha = (ipha_t *)mp->b_rptr;
2831     bzero(&ixas, sizeof(ixas));
2832     ixas.ixas_flags = IXAF_BASIC_SIMPLE_V4;
2833     ixas.ixas_zoneid = ira->ira_zoneid;

```

```

2834     ixas.ixas_ifindex = 0;
2835     ixas.ixas_ipst = ipst;
2836     ixas.ixas_cred = kcred;
2837     ixas.ixas_cpuid = NOPID;
2838     ixas.ixas_tsl = ira->ira_tsl; /* Behave as a multi-level responder */
2839     ixas.ixas_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
2841     if (ira->ira_flags & IRAF_IPSEC_SECURE) {
2842         /*
2843          * Apply IPsec based on how IPsec was applied to
2844          * the packet that had the error.
2845          *
2846          * If it was an outbound packet that caused the ICMP
2847          * error, then the caller will have setup the IRA
2848          * appropriately.
2849          */
2850         if (!ipsec_in_to_out(ira, &ixas, mp, ipha, NULL)) {
2851             BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
2852             /* Note: mp already consumed and ip_drop_packet done */
2853             return;
2854         }
2855     } else {
2856         /*
2857          * This is in clear. The icmp message we are building
2858          * here should go out in clear, independent of our policy.
2859          */
2860         ixas.ixas_flags |= IXAF_NO_IPSEC;
2861     }
2863     /* Remember our eventual destination */
2864     dst = ipha->ipha_src;
2866     /*
2867     * If the packet was for one of our unicast addresses, make
2868     * sure we respond with that as the source. Otherwise
2869     * have ip_output_simple pick the source address.
2870     */
2871     ire = ire_fhtable_lookup_v4(ipha->ipha_dst, 0, 0,
2872 (IRE_LOCAL|IRE_LOOPBACK), NULL, ira->ira_zoneid, NULL,
2873 MATCH_IRE_TYPE|MATCH_IRE_ZONEONLY, 0, ipst, NULL);
2874     if (ire != NULL) {
2875         ire_refrele(ire);
2876         src = ipha->ipha_dst;
2877     } else {
2878         src = INADDR_ANY;
2879         ixas.ixas_flags |= IXAF_SET_SOURCE;
2880     }
2882     /*
2883     * Check if we can send back more than 8 bytes in addition to
2884     * the IP header. We try to send 64 bytes of data and the internal
2885     * header in the special cases of ipv4 encapsulated ipv4 or ipv6.
2886     */
2887     len_needed = IPH_HDR_LENGTH(ipha);
2888     if (ipha->ipha_protocol == IPPROTO_ENCAP ||
2889         ipha->ipha_protocol == IPPROTO_IPV6) {
2890         if (!pullupmsg(mp, -1)) {
2891             BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutDiscards);
2892             ip_drop_output("ipIfStatsOutDiscards", mp, NULL);
2893             freemsg(mp);
2894             return;
2895         }
2896         ipha = (ipha_t *)mp->b_rptr;
2898         if (ipha->ipha_protocol == IPPROTO_ENCAP) {
2899             len_needed += IPH_HDR_LENGTH(((uchar_t *)ipha +

```

```

2900         len_needed));
2901     } else {
2902         ip6_t *ip6h = (ip6_t *)((uchar_t *)ipha + len_needed);
2903
2904         ASSERT(ipha->ipha_protocol == IPPROTO_IPV6);
2905         len_needed += ip_hdr_length_v6(mp, ip6h);
2906     }
2907 }
2908 len_needed += ipst->ips_ip_icmp_return;
2909 msg_len = msgdsize(mp);
2910 if (msg_len > len_needed) {
2911     (void) adjmsg(mp, len_needed - msg_len);
2912     msg_len = len_needed;
2913 }
2914 mp1 = allocb(sizeof (icmp_ipha) + len, BPRI_MED);
2915 if (mp1 == NULL) {
2916     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutErrors);
2917     freemsg(mp);
2918     return;
2919 }
2920 mp1->b_cont = mp;
2921 mp = mp1;
2922
2923 /*
2924  * Set IXAF_TRUSTED_ICMP so we can let the ICMP messages this
2925  * node generates be accepted in peace by all on-host destinations.
2926  * If we do NOT assume that all on-host destinations trust
2927  * self-generated ICMP messages, then rework here, ip6.c, and spd.c.
2928  * (Look for IXAF_TRUSTED_ICMP).
2929  */
2930 ixas.ixaf_flags |= IXAF_TRUSTED_ICMP;
2931
2932 ipha = (ipha_t *)mp->b_rptr;
2933 mp1->b_wptr = (uchar_t *)ipha + (sizeof (icmp_ipha) + len);
2934 *ipha = icmp_ipha;
2935 ipha->ipha_src = src;
2936 ipha->ipha_dst = dst;
2937 ipha->ipha_ttl = ipst->ips_ip_def_ttl;
2938 msg_len += sizeof (icmp_ipha) + len;
2939 if (msg_len > IP_MAXPACKET) {
2940     (void) adjmsg(mp, IP_MAXPACKET - msg_len);
2941     msg_len = IP_MAXPACKET;
2942 }
2943 ipha->ipha_length = htons((uint16_t)msg_len);
2944 icmph = (icmph_t *)&ipha[1];
2945 bcopy(stuff, icmph, len);
2946 icmph->icmph_checksum = 0;
2947 icmph->icmph_checksum = IP_CSUM(mp, (int32_t)sizeof (ipha_t), 0);
2948 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutMsgs);
2949
2950 (void) ip_output_simple(mp, &ixas);
2951 ixa_cleanup(&ixas);
2952 }
2953
2954 /*
2955  * Determine if an ICMP error packet can be sent given the rate limit.
2956  * The limit consists of an average frequency (icmp_pkt_err_interval measured
2957  * in milliseconds) and a burst size. Burst size number of packets can
2958  * be sent arbitrarily closely spaced.
2959  * The state is tracked using two variables to implement an approximate
2960  * token bucket filter:
2961  *   icmp_pkt_err_last - lbolt value when the last burst started
2962  *   icmp_pkt_err_sent - number of packets sent in current burst
2963  */
2964 boolean_t
2965 icmp_err_rate_limit(ip_stack_t *ipst)

```

```

2966 {
2967     clock_t now = TICK_TO_MSEC(ddi_get_lbolt());
2968     uint_t refilled; /* Number of packets refilled in tbf since last */
2969     /* Guard against changes by loading into local variable */
2970     uint_t err_interval = ipst->ips_ip_icmp_err_interval;
2971
2972     if (err_interval == 0)
2973         return (B_FALSE);
2974
2975     if (ipst->ips_icmp_pkt_err_last > now) {
2976         /* 100HZ lbolt in ms for 32bit arch wraps every 49.7 days */
2977         ipst->ips_icmp_pkt_err_last = 0;
2978         ipst->ips_icmp_pkt_err_sent = 0;
2979     }
2980     /*
2981      * If we are in a burst update the token bucket filter.
2982      * Update the "last" time to be close to "now" but make sure
2983      * we don't lose precision.
2984      */
2985     if (ipst->ips_icmp_pkt_err_sent != 0) {
2986         refilled = (now - ipst->ips_icmp_pkt_err_last) / err_interval;
2987         if (refilled > ipst->ips_icmp_pkt_err_sent) {
2988             ipst->ips_icmp_pkt_err_sent = 0;
2989         } else {
2990             ipst->ips_icmp_pkt_err_sent -= refilled;
2991             ipst->ips_icmp_pkt_err_last += refilled * err_interval;
2992         }
2993     }
2994     if (ipst->ips_icmp_pkt_err_sent == 0) {
2995         /* Start of new burst */
2996         ipst->ips_icmp_pkt_err_last = now;
2997     }
2998     if (ipst->ips_icmp_pkt_err_sent < ipst->ips_ip_icmp_err_burst) {
2999         ipst->ips_icmp_pkt_err_sent++;
3000         ipldbg(("icmp_err_rate_limit: %d sent in burst\n",
3001             ipst->ips_icmp_pkt_err_sent));
3002         return (B_FALSE);
3003     }
3004     ipldbg(("icmp_err_rate_limit: dropped\n"));
3005     return (B_TRUE);
3006 }
3007
3008 /*
3009  * Check if it is ok to send an IPv4 ICMP error packet in
3010  * response to the IPv4 packet in mp.
3011  * Free the message and return null if no
3012  * ICMP error packet should be sent.
3013  */
3014 static mblk_t *
3015 icmp_pkt_err_ok(mblk_t *mp, ip_recv_attr_t *ira)
3016 {
3017     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
3018     icmph_t *icmph;
3019     ipha_t *ipha;
3020     uint_t len_needed;
3021
3022     if (!mp)
3023         return (NULL);
3024     ipha = (ipha_t *)mp->b_rptr;
3025     if (ip_csum_hdr(ipha)) {
3026         BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsInCksumErrs);
3027         ip_drop_input("ipIfStatsInCksumErrs", mp, NULL);
3028         freemsg(mp);
3029         return (NULL);
3030     }
3031     if (ip_type_v4(ipha->ipha_dst, ipst) == IRE_BROADCAST ||

```



```

3032     ip_type_v4(ipha->ipha_src, ipst) == IRE_BROADCAST ||
3033     CLASSD(ipha->ipha_dst) ||
3034     CLASSD(ipha->ipha_src) ||
3035     (ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_OFFSET)) {
3036     /* Note: only errors to the fragment with offset 0 */
3037     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
3038     freemsg(mp);
3039     return (NULL);
3040 }
3041 if (ipha->ipha_protocol == IPPROTO_ICMP) {
3042     /*
3043     * Check the ICMP type. RFC 1122 sez: don't send ICMP
3044     * errors in response to any ICMP errors.
3045     */
3046     len_needed = IPH_HDR_LENGTH(ipha) + ICMPH_SIZE;
3047     if (mp->b_wptr - mp->b_rptr < len_needed) {
3048         if (!pullupmsg(mp, len_needed)) {
3049             BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
3050             freemsg(mp);
3051             return (NULL);
3052         }
3053         ipha = (ipha_t *)mp->b_rptr;
3054     }
3055     icmp_h = (icmp_h_t *)
3056         (&((char *)ipha)[IPH_HDR_LENGTH(ipha)]);
3057     switch (icmp_h->icmp_type) {
3058     case ICMP_DEST_UNREACHABLE:
3059     case ICMP_SOURCE_QUENCH:
3060     case ICMP_TIME_EXCEEDED:
3061     case ICMP_PARAM_PROBLEM:
3062     case ICMP_REDIRECT:
3063         BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
3064         freemsg(mp);
3065         return (NULL);
3066     default:
3067         break;
3068     }
3069 }
3070 /*
3071 * If this is a labeled system, then check to see if we're allowed to
3072 * send a response to this particular sender. If not, then just drop.
3073 */
3074 if (is_system_labeled() && !tsol_can_reply_error(mp, ira)) {
3075     ip2dbg(("icmp_pkt_err_ok: can't respond to packet\n"));
3076     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
3077     freemsg(mp);
3078     return (NULL);
3079 }
3080 if (icmp_err_rate_limit(ipst)) {
3081     /*
3082     * Only send ICMP error packets every so often.
3083     * This should be done on a per port/source basis,
3084     * but for now this will suffice.
3085     */
3086     freemsg(mp);
3087     return (NULL);
3088 }
3089 return (mp);
3090 }
3092 /*
3093 * Called when a packet was sent out the same link that it arrived on.
3094 * Check if it is ok to send a redirect and then send it.
3095 */
3096 void
3097 ip_send_potential_redirect_v4(mblk_t *mp, ipha_t *ipha, ire_t *ire,

```

```

3098     ip_rcv_attr_t *ira)
3099 {
3100     ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
3101     ip_addr_t       src, nhop;
3102     mblk_t          *mpl;
3103     ire_t           *nhop_ire;
3104
3105     /*
3106     * Check the source address to see if it originated
3107     * on the same logical subnet it is going back out on.
3108     * If so, we should be able to send it a redirect.
3109     * Avoid sending a redirect if the destination
3110     * is directly connected (i.e., we matched an IRE_ONLINK),
3111     * or if the packet was source routed out this interface.
3112     */
3113     /* We avoid sending a redirect if the
3114     * destination is directly connected
3115     * because it is possible that multiple
3116     * IP subnets may have been configured on
3117     * the link, and the source may not
3118     * be on the same subnet as ip destination,
3119     * even though they are on the same
3120     * physical link.
3121     */
3122     if ((ire->ire_type & IRE_ONLINK) ||
3123         ip_source_routed(ipha, ipst))
3124         return;
3126     nhop_ire = ire_nexthop(ire);
3127     if (nhop_ire == NULL)
3128         return;
3130     nhop = nhop_ire->ire_addr;
3132     if (nhop_ire->ire_type & IRE_IF_CLONE) {
3133         ire_t *ire2;
3135         /* Follow ire_dep_parent to find non-clone IRE_INTERFACE */
3136         mutex_enter(&nhop_ire->ire_lock);
3137         ire2 = nhop_ire->ire_dep_parent;
3138         if (ire2 != NULL)
3139             ire_refhold(ire2);
3140         mutex_exit(&nhop_ire->ire_lock);
3141         ire_refrele(nhop_ire);
3142         nhop_ire = ire2;
3143     }
3144     if (nhop_ire == NULL)
3145         return;
3147     ASSERT(!(nhop_ire->ire_type & IRE_IF_CLONE));
3149     src = ipha->ipha_src;
3151     /*
3152     * We look at the interface ire for the nexthop,
3153     * to see if ipha_src is in the same subnet
3154     * as the nexthop.
3155     */
3156     if ((src & nhop_ire->ire_mask) == (nhop & nhop_ire->ire_mask)) {
3157         /*
3158         * The source is directly connected.
3159         */
3160         mpl = copymsg(mp);
3161         if (mpl != NULL) {
3162             icmp_send_redirect(mpl, nhop, ira);
3163         }

```

```

3164     }
3165     ire_refrele(nhop_ire);
3166 }

3168 /*
3169  * Generate an ICMP redirect message.
3170  */
3171 static void
3172 icmp_send_redirect(mblk_t *mp, ipaddr_t gateway, ip_recv_attr_t *ira)
3173 {
3174     icmph_t icmph;
3175     ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3177     mp = icmp_pkt_err_ok(mp, ira);
3178     if (mp == NULL)
3179         return;

3181     bzero(&icmph, sizeof (icmph_t));
3182     icmph.icmph_type = ICMP_REDIRECT;
3183     icmph.icmph_code = 1;
3184     icmph.icmph_rd_gateway = gateway;
3185     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutRedirects);
3186     icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3187 }

3189 /*
3190  * Generate an ICMP time exceeded message.
3191  */
3192 void
3193 icmp_time_exceeded(mblk_t *mp, uint8_t code, ip_recv_attr_t *ira)
3194 {
3195     icmph_t icmph;
3196     ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3198     mp = icmp_pkt_err_ok(mp, ira);
3199     if (mp == NULL)
3200         return;

3202     bzero(&icmph, sizeof (icmph_t));
3203     icmph.icmph_type = ICMP_TIME_EXCEEDED;
3204     icmph.icmph_code = code;
3205     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimeExcds);
3206     icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3207 }

3209 /*
3210  * Generate an ICMP unreachable message.
3211  * When called from ip_output side a minimal ip_recv_attr_t needs to be
3212  * constructed by the caller.
3213  */
3214 void
3215 icmp_unreachable(mblk_t *mp, uint8_t code, ip_recv_attr_t *ira)
3216 {
3217     icmph_t icmph;
3218     ip_stack_t *ipst = ira->ira_ill->ill_ipst;

3220     mp = icmp_pkt_err_ok(mp, ira);
3221     if (mp == NULL)
3222         return;

3224     bzero(&icmph, sizeof (icmph_t));
3225     icmph.icmph_type = ICMP_DEST_UNREACHABLE;
3226     icmph.icmph_code = code;
3227     BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDestUnreachs);
3228     icmp_pkt(mp, &icmph, sizeof (icmph_t), ira);
3229 }

```

```

3231 /*
3232  * Latch in the IPsec state for a stream based the policy in the listener
3233  * and the actions in the ip_recv_attr_t.
3234  * Called directly from TCP and SCTP.
3235  */
3236 boolean_t
3237 ip_ipsec_policy_inherit(conn_t *connp, conn_t *lconnp, ip_recv_attr_t *ira)
3238 {
3239     ASSERT(lconnp->conn_policy != NULL);
3240     ASSERT(connp->conn_policy == NULL);

3242     IPPH_REFHOLD(lconnp->conn_policy);
3243     connp->conn_policy = lconnp->conn_policy;

3245     if (ira->ira_ipsec_action != NULL) {
3246         if (connp->conn_latch == NULL) {
3247             connp->conn_latch = iplatch_create();
3248             if (connp->conn_latch == NULL)
3249                 return (B_FALSE);
3250         }
3251         ipsec_latch_inbound(connp, ira);
3252     }
3253     return (B_TRUE);
3254 }

3256 /*
3257  * Verify whether or not the IP address is a valid local address.
3258  * Could be a unicast, including one for a down interface.
3259  * If allow_mcbc then a multicast or broadcast address is also
3260  * acceptable.
3261  *
3262  * In the case of a broadcast/multicast address, however, the
3263  * upper protocol is expected to reset the src address
3264  * to zero when we return IPVL_MCAST/IPVL_BCAST so that
3265  * no packets are emitted with broadcast/multicast address as
3266  * source address (that violates hosts requirements RFC 1122)
3267  * The addresses valid for bind are:
3268  * (1) - INADDR_ANY (0)
3269  * (2) - IP address of an UP interface
3270  * (3) - IP address of a DOWN interface
3271  * (4) - valid local IP broadcast addresses. In this case
3272  * the conn will only receive packets destined to
3273  * the specified broadcast address.
3274  * (5) - a multicast address. In this case
3275  * the conn will only receive packets destined to
3276  * the specified multicast address. Note: the
3277  * application still has to issue an
3278  * IP_ADD_MEMBERSHIP socket option.
3279  *
3280  * In all the above cases, the bound address must be valid in the current zone.
3281  * When the address is loopback, multicast or broadcast, there might be many
3282  * matching IREs so bind has to look up based on the zone.
3283  */
3284 ip_laddr_t
3285 ip_laddr_verify_v4(ipaddr_t src_addr, zoneid_t zoneid,
3286     ip_stack_t *ipst, boolean_t allow_mcbc)
3287 {
3288     ire_t *src_ire;

3290     ASSERT(src_addr != INADDR_ANY);

3292     src_ire = ire_ftable_lookup_v4(src_addr, 0, 0, 0,
3293         NULL, zoneid, NULL, MATCH_IRE_ZONEONLY, 0, ipst, NULL);

3295     /*

```

```

3296     * If an address other than in6addr_any is requested,
3297     * we verify that it is a valid address for bind
3298     * Note: Following code is in if-else-if form for
3299     * readability compared to a condition check.
3300     */
3301     if (src_ire != NULL && (src_ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK))) {
3302         /*
3303          * (2) Bind to address of local UP interface
3304          */
3305         ire_refrele(src_ire);
3306         return (IPVL_UNICAST_UP);
3307     } else if (src_ire != NULL && src_ire->ire_type & IRE_BROADCAST) {
3308         /*
3309          * (4) Bind to broadcast address
3310          */
3311         ire_refrele(src_ire);
3312         if (allow_mcbc)
3313             return (IPVL_BCAST);
3314         else
3315             return (IPVL_BAD);
3316     } else if (CLASSD(src_addr)) {
3317         /* (5) bind to multicast address. */
3318         if (src_ire != NULL)
3319             ire_refrele(src_ire);
3320
3321         if (allow_mcbc)
3322             return (IPVL_MCAST);
3323         else
3324             return (IPVL_BAD);
3325     } else {
3326         ipif_t *ipif;
3327
3328         /*
3329          * (3) Bind to address of local DOWN interface?
3330          * (ipif_lookup_addr() looks up all interfaces
3331          * but we do not get here for UP interfaces
3332          * - case (2) above)
3333          */
3334         if (src_ire != NULL)
3335             ire_refrele(src_ire);
3336
3337         ipif = ipif_lookup_addr(src_addr, NULL, zoneid, ipst);
3338         if (ipif == NULL)
3339             return (IPVL_BAD);
3340
3341         /* Not a useful source? */
3342         if (ipif->ipif_flags & (IPIF_NOLOCAL | IPIF_ANYCAST)) {
3343             ipif_refrele(ipif);
3344             return (IPVL_BAD);
3345         }
3346         ipif_refrele(ipif);
3347         return (IPVL_UNICAST_DOWN);
3348     }
3349 }
3350
3351 /*
3352  * Insert in the bind fanout for IPv4 and IPv6.
3353  * The caller should already have used ip_laddr_verify_v*() before calling
3354  * this.
3355  */
3356 int
3357 ip_laddr_fanout_insert(conn_t *connp)
3358 {
3359     int            error;
3360
3361     /*

```

```

3362     * Allow setting new policies. For example, disconnects result
3363     * in us being called. As we would have set conn_policy_cached
3364     * to B_TRUE before, we should set it to B_FALSE, so that policy
3365     * can change after the disconnect.
3366     */
3367     connp->conn_policy_cached = B_FALSE;
3368
3369     error = ipcl_bind_insert(connp);
3370     if (error != 0) {
3371         if (connp->conn_anon_port) {
3372             (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
3373                 connp->conn_mlp_type, connp->conn_proto,
3374                 ntohs(connp->conn_lport), B_FALSE);
3375         }
3376         connp->conn_mlp_type = mlptSingle;
3377     }
3378     return (error);
3379 }
3380
3381 /*
3382  * Verify that both the source and destination addresses are valid. If
3383  * IPDF_VERIFY_DST is not set, then the destination address may be unreachable,
3384  * i.e. have no route to it. Protocols like TCP want to verify destination
3385  * reachability, while tunnels do not.
3386  *
3387  * Determine the route, the interface, and (optionally) the source address
3388  * to use to reach a given destination.
3389  * Note that we allow connect to broadcast and multicast addresses when
3390  * IPDF_ALLOW_MCBC is set.
3391  * first_hop and dst_addr are normally the same, but if source routing
3392  * they will differ; in that case the first_hop is what we'll use for the
3393  * routing lookup but the dce and label checks will be done on dst_addr,
3394  *
3395  * If uinfo is set, then we fill in the best available information
3396  * we have for the destination. This is based on (in priority order) any
3397  * metrics and path MTU stored in a dce_t, route metrics, and finally the
3398  * ill_mtu/ill_mc_mtu.
3399  *
3400  * Tsol note: If we have a source route then dst_addr != firsthop. But we
3401  * always do the label check on dst_addr.
3402  */
3403 int
3404 ip_set_destination_v4(ipaddr_t *src_addrp, ipaddr_t dst_addr, ipaddr_t firsthop,
3405     ip_xmit_attr_t *ixa, iulp_t *uinfo, uint32_t flags, uint_t mac_mode)
3406 {
3407     ire_t          *ire = NULL;
3408     int             error = 0;
3409     ipaddr_t        setsrc;
3410     zoneid_t        zoneid = ix->ixa_zoneid; /* RTF_SETSRC */
3411     ip_stack_t      *ipst = ix->ixa_ipst; /* Honors SO_ALLZONES */
3412     dce_t           *dce;
3413     uint_t          pmtu;
3414     uint_t          generation;
3415     nce_t           *nce;
3416     ill_t           *ill = NULL;
3417     boolean_t       multirt = B_FALSE;
3418
3419     ASSERT(ix->ixa_flags & IXAF_IS_IPV4);
3420
3421     /*
3422      * We never send to zero; the ULPs map it to the loopback address.
3423      * We can't allow it since we use zero to mean uninitialized in some
3424      * places.
3425      */
3426     ASSERT(dst_addr != INADDR_ANY);

```

```

3428     if (is_system_labeled()) {
3429         ts_label_t *tsl = NULL;

3431         error = tsol_check_dest(ixa->ixa_tsl, &dst_addr, IPV4_VERSION,
3432             mac_mode, (flags & IPDF_ZONE_IS_GLOBAL) != 0, &tsl);
3433         if (error != 0)
3434             return (error);
3435         if (tsl != NULL) {
3436             /* Update the label */
3437             ip_xmit_attr_replace_tsl(ixa, tsl);
3438         }
3439     }

3441     setsrc = INADDR_ANY;
3442     /*
3443      * Select a route; For IPMP interfaces, we would only select
3444      * a "hidden" route (i.e., going through a specific under_ill)
3445      * if ixa_ifindex has been specified.
3446      */
3447     ire = ip_select_route_v4(firsthop, *src_addrp, ixa,
3448         &generation, &setsrc, &error, &multirt);
3449     ASSERT(ire != NULL); /* IRE_NOROUTE if none found */
3450     if (error != 0)
3451         goto bad_addr;

3453     /*
3454      * ire can't be a broadcast or multicast unless IPDF_ALLOW_MCBC is set.
3455      * If IPDF_VERIFY_DST is set, the destination must be reachable;
3456      * Otherwise the destination needn't be reachable.
3457      *
3458      * If we match on a reject or black hole, then we've got a
3459      * local failure. May as well fail out the connect() attempt,
3460      * since it's never going to succeed.
3461      */
3462     if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
3463         /*
3464          * If we're verifying destination reachability, we always want
3465          * to complain here.
3466          *
3467          * If we're not verifying destination reachability but the
3468          * destination has a route, we still want to fail on the
3469          * temporary address and broadcast address tests.
3470          *
3471          * In both cases do we let the code continue so some reasonable
3472          * information is returned to the caller. That enables the
3473          * caller to use (and even cache) the IRE. conn_ip_output will
3474          * use the generation mismatch path to check for the unreachable
3475          * case thereby avoiding any specific check in the main path.
3476          */
3477         ASSERT(generation == IRE_GENERATION_VERIFY);
3478         if (flags & IPDF_VERIFY_DST) {
3479             /*
3480              * Set errno but continue to set up ixa_ire to be
3481              * the RTF_REJECT|RTF_BLACKHOLE IRE.
3482              * That allows callers to use ip_output to get an
3483              * ICMP error back.
3484              */
3485             if (!(ire->ire_type & IRE_HOST))
3486                 error = ENETUNREACH;
3487             else
3488                 error = EHOSTUNREACH;
3489         }
3490     }

3492     if ((ire->ire_type & (IRE_BROADCAST|IRE_MULTICAST)) &&
3493         !(flags & IPDF_ALLOW_MCBC)) {

```

```

3494         ire_refrele(ire);
3495         ire = ire_reject(ipst, B_FALSE);
3496         generation = IRE_GENERATION_VERIFY;
3497         error = ENETUNREACH;
3498     }

3500     /* Cache things */
3501     if (ixa->ixa_ire != NULL)
3502         ire_refrele_notr(ixa->ixa_ire);
3503     #ifdef DEBUG
3504         ire_refhold_notr(ire);
3505         ire_refrele(ire);
3506     #endif
3507     ixa->ixa_ire = ire;
3508     ixa->ixa_ire_generation = generation;

3510     /*
3511      * Ensure that ixa_dce is always set any time that ixa_ire is set,
3512      * since some callers will send a packet to conn_ip_output() even if
3513      * there's an error.
3514      */
3515     if (flags & IPDF_UNIQUE_DCE) {
3516         /* Fallback to the default dce if allocation fails */
3517         dce = dce_lookup_and_add_v4(dst_addr, ipst);
3518         if (dce != NULL)
3519             generation = dce->dce_generation;
3520         else
3521             dce = dce_lookup_v4(dst_addr, ipst, &generation);
3522     } else {
3523         dce = dce_lookup_v4(dst_addr, ipst, &generation);
3524     }
3525     ASSERT(dce != NULL);
3526     if (ixa->ixa_dce != NULL)
3527         dce_refrele_notr(ixa->ixa_dce);
3528     #ifdef DEBUG
3529         dce_refhold_notr(dce);
3530         dce_refrele(dce);
3531     #endif
3532     ixa->ixa_dce = dce;
3533     ixa->ixa_dce_generation = generation;

3535     /*
3536      * For multicast with multirt we have a flag passed back from
3537      * ire_lookup_multi_ill_v4 since we don't have an IRE for each
3538      * possible multicast address.
3539      * We also need a flag for multicast since we can't check
3540      * whether RTF_MULTIRT is set in ixa_ire for multicast.
3541      */
3542     if (multirt) {
3543         ixa->ixa_postfragfn = ip_postfrag_multirt_v4;
3544         ixa->ixa_flags |= IXAF_MULTIRT_MULTICAST;
3545     } else {
3546         ixa->ixa_postfragfn = ire->ire_postfragfn;
3547         ixa->ixa_flags &= ~IXAF_MULTIRT_MULTICAST;
3548     }
3549     if (!(ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))) {
3550         /* Get an nce to cache. */
3551         nce = ire_to_nce(ire, firsthop, NULL);
3552         if (nce == NULL) {
3553             /* Allocation failure? */
3554             ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3555         } else {
3556             if (ixa->ixa_nce != NULL)
3557                 nce_refrele(ixa->ixa_nce);
3558             ixa->ixa_nce = nce;
3559         }

```

```

3560     }
3561
3562     /*
3563     * If the source address is a loopback address, the
3564     * destination had best be local or multicast.
3565     * If we are sending to an IRE_LOCAL using a loopback source then
3566     * it had better be the same zoneid.
3567     */
3568     if (*src_addrp == htonl(INADDR_LOOPBACK)) {
3569         if ((ire->ire_type & IRE_LOCAL) && ire->ire_zoneid != zoneid) {
3570             ire = NULL; /* Stored in ixa_ire */
3571             error = EADDRNOTAVAIL;
3572             goto bad_addr;
3573         }
3574         if (!(ire->ire_type & (IRE_LOOPBACK|IRE_LOCAL|IRE_MULTICAST))) {
3575             ire = NULL; /* Stored in ixa_ire */
3576             error = EADDRNOTAVAIL;
3577             goto bad_addr;
3578         }
3579     }
3580     if (ire->ire_type & IRE_BROADCAST) {
3581         /*
3582         * If the ULP didn't have a specified source, then we
3583         * make sure we reselect the source when sending
3584         * broadcasts out different interfaces.
3585         */
3586         if (flags & IPDF_SELECT_SRC)
3587             ixa->ixa_flags |= IXAF_SET_SOURCE;
3588         else
3589             ixa->ixa_flags &= ~IXAF_SET_SOURCE;
3590     }
3591
3592     /*
3593     * Does the caller want us to pick a source address?
3594     */
3595     if (flags & IPDF_SELECT_SRC) {
3596         ipaddr_t      src_addr;
3597
3598         /*
3599         * We use use ire_nexthop_ill to avoid the under ipmp
3600         * interface for source address selection. Note that for ipmp
3601         * probe packets, ixa_ifindex would have been specified, and
3602         * the ip_select_route() invocation would have picked an ire
3603         * will ire_ill pointing at an under interface.
3604         */
3605         ill = ire_nexthop_ill(ire);
3606
3607         /* If unreachable we have no ill but need some source */
3608         if (ill == NULL) {
3609             src_addr = htonl(INADDR_LOOPBACK);
3610             /* Make sure we look for a better source address */
3611             generation = SRC_GENERATION_VERIFY;
3612         } else {
3613             error = ip_select_source_v4(ill, setsrc, dst_addr,
3614             ixa->ixa_multicast_ifaddr, zoneid,
3615             ipst, &src_addr, &generation, NULL);
3616             if (error != 0) {
3617                 ire = NULL; /* Stored in ixa_ire */
3618                 goto bad_addr;
3619             }
3620         }
3621     }
3622
3623     /*
3624     * We allow the source address to to down.
3625     * However, we check that we don't use the loopback address
3626     * as a source when sending out on the wire.

```

```

3626     */
3627     if ((src_addr == htonl(INADDR_LOOPBACK)) &&
3628         !(ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK|IRE_MULTICAST)) &&
3629         !(ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))) {
3630         ire = NULL; /* Stored in ixa_ire */
3631         error = EADDRNOTAVAIL;
3632         goto bad_addr;
3633     }
3634
3635     *src_addrp = src_addr;
3636     ixa->ixa_src_generation = generation;
3637 }
3638
3639 /*
3640 * Make sure we don't leave an unreachable ixa_nce in place
3641 * since ip_select_route is used when we unplumb i.e., remove
3642 * references on ixa_ire, ixa_nce, and ixa_dce.
3643 */
3644 nce = ixa->ixa_nce;
3645 if (nce != NULL && nce->nce_is_condemned) {
3646     nce_refrele(nce);
3647     ixa->ixa_nce = NULL;
3648     ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3649 }
3650
3651 /*
3652 * The caller has set IXAF_PMTU_DISCOVERY if path MTU is desired.
3653 * However, we can't do it for IPv4 multicast or broadcast.
3654 */
3655 if (ire->ire_type & (IRE_BROADCAST|IRE_MULTICAST))
3656     ixa->ixa_flags &= ~IXAF_PMTU_DISCOVERY;
3657
3658 /*
3659 * Set initial value for fragmentation limit. Either conn_ip_output
3660 * or ULP might updates it when there are routing changes.
3661 * Handles a NULL ixa_ire->ire_ill or a NULL ixa_nce for RTF_REJECT.
3662 */
3663 pmtu = ip_get_pmtu(ixa);
3664 ixa->ixa_fragsize = pmtu;
3665 /* Make sure ixa_fragsize and ixa_pmtu remain identical */
3666 if (ixa->ixa_flags & IXAF_VERIFY_PMTU)
3667     ixa->ixa_pmtu = pmtu;
3668
3669 /*
3670 * Extract information useful for some transports.
3671 * First we look for DCE metrics. Then we take what we have in
3672 * the metrics in the route, where the offlink is used if we have
3673 * one.
3674 */
3675 if (uinfo != NULL) {
3676     bzero(uinfo, sizeof (*uinfo));
3677
3678     if (dce->dce_flags & DCEF_UINFO)
3679         *uinfo = dce->dce_uinfo;
3680
3681     rts_merge_metrics(uinfo, &ire->ire_metrics);
3682
3683     /* Allow ire_metrics to decrease the path MTU from above */
3684     if (uinfo->iulp_mtu == 0 || uinfo->iulp_mtu > pmtu)
3685         uinfo->iulp_mtu = pmtu;
3686
3687     uinfo->iulp_localnet = (ire->ire_type & IRE_ONLINK) != 0;
3688     uinfo->iulp_loopback = (ire->ire_type & IRE_LOOPBACK) != 0;
3689     uinfo->iulp_local = (ire->ire_type & IRE_LOCAL) != 0;
3690 }

```

```

3692     if (ill != NULL)
3693         ill_refrele(ill);

3695     return (error);

3697 bad_addr:
3698     if (ire != NULL)
3699         ire_refrele(ire);

3701     if (ill != NULL)
3702         ill_refrele(ill);

3704     /*
3705      * Make sure we don't leave an unreachable ixa_nce in place
3706      * since ip_select_route is used when we unplumb i.e., remove
3707      * references on ixa_ire, ixa_nce, and ixa_dce.
3708      */
3709     nce = ixa->ixa_nce;
3710     if (nce != NULL && nce->nce_is_condemned) {
3711         nce_refrele(nce);
3712         ixa->ixa_nce = NULL;
3713         ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
3714     }

3716     return (error);
3717 }

3720 /*
3721  * Get the base MTU for the case when path MTU discovery is not used.
3722  * Takes the MTU of the IRE into account.
3723  */
3724 uint_t
3725 ip_get_base_mtu(ill_t *ill, ire_t *ire)
3726 {
3727     uint_t mtu;
3728     uint_t iremtu = ire->ire_metrics.iulp_mtu;

3730     if (ire->ire_type & (IRE_MULTICAST|IRE_BROADCAST))
3731         mtu = ill->ill_mc_mtu;
3732     else
3733         mtu = ill->ill_mtu;

3735     if (iremtu != 0 && iremtu < mtu)
3736         mtu = iremtu;

3738     return (mtu);
3739 }

3741 /*
3742  * Get the PMTU for the attributes. Handles both IPv4 and IPv6.
3743  * Assumes that ixa_ire, dce, and nce have already been set up.
3744  * The caller has set IXAF_PMTU_DISCOVERY if path MTU discovery is desired.
3745  * We avoid path MTU discovery if it is disabled with ndd.
3746  * Furthermore, if the path MTU is too small, then we don't set DF for IPv4.
3747  * NOTE: We also used to turn it off for source routed packets. That
3748  * is no longer required since the dce is per final destination.
3749  */
3750 uint_t
3751 ip_get_pmtu(ip_xmit_attr_t *ixa)
3752 {
3753     ip_stack_t      *ipst = ixa->ixa_ipst;
3754     dce_t            *dce;
3755     nce_t            *nce;

```

```

3758     ire_t            *ire;
3759     uint_t           pmtu;

3761     ire = ixa->ixa_ire;
3762     dce = ixa->ixa_dce;
3763     nce = ixa->ixa_nce;

3765     /*
3766      * If path MTU discovery has been turned off by ndd, then we ignore
3767      * any dce_pmtu and for IPv4 we will not set DF.
3768      */
3769     if (!ipst->ips_ip_path_mtu_discovery)
3770         ixa->ixa_flags &= ~IXAF_PMTU_DISCOVERY;

3772     pmtu = IP_MAXPACKET;
3773     /*
3774      * Decide whether whether IPv4 sets DF
3775      * For IPv6 "no DF" means to use the 1280 mtu
3776      */
3777     if (ixa->ixa_flags & IXAF_PMTU_DISCOVERY) {
3778         ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;
3779     } else {
3780         ixa->ixa_flags &= ~IXAF_PMTU_IPV4_DF;
3781         if (!(ixa->ixa_flags & IXAF_IS_IPV4))
3782             pmtu = IPV6_MIN_MTU;
3783     }

3785     /* Check if the PMTU is too old before we use it */
3786     if ((dce->dce_flags & DCEF_PMTU) &&
3787         TICK_TO_SEC(ddi_get_lbolt64()) - dce->dce_last_change_time >
3788         ipst->ips_ip_pathmtu_interval) {
3789         /*
3790          * Older than 20 minutes. Drop the path MTU information.
3791          */
3792         mutex_enter(&dce->dce_lock);
3793         dce->dce_flags &= ~(DCEF_PMTU|DCEF_TOO_SMALL_PMTU);
3794         dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
3795         mutex_exit(&dce->dce_lock);
3796         dce_increment_generation(dce);
3797     }

3799     /* The metrics on the route can lower the path MTU */
3800     if (ire->ire_metrics.iulp_mtu != 0 &&
3801         ire->ire_metrics.iulp_mtu < pmtu)
3802         pmtu = ire->ire_metrics.iulp_mtu;

3804     /*
3805      * If the path MTU is smaller than some minimum, we still use dce_pmtu
3806      * above (would be 576 for IPv4 and 1280 for IPv6), but we clear
3807      * IXAF_PMTU_IPV4_DF so that we avoid setting DF for IPv4.
3808      */
3809     if (ixa->ixa_flags & IXAF_PMTU_DISCOVERY) {
3810         if (dce->dce_flags & DCEF_PMTU) {
3811             if (dce->dce_pmtu < pmtu)
3812                 pmtu = dce->dce_pmtu;

3814             if (dce->dce_flags & DCEF_TOO_SMALL_PMTU) {
3815                 ixa->ixa_flags |= IXAF_PMTU_TOO_SMALL;
3816                 ixa->ixa_flags &= ~IXAF_PMTU_IPV4_DF;
3817             } else {
3818                 ixa->ixa_flags &= ~IXAF_PMTU_TOO_SMALL;
3819                 ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;
3820             }
3821         } else {
3822             ixa->ixa_flags &= ~IXAF_PMTU_TOO_SMALL;
3823             ixa->ixa_flags |= IXAF_PMTU_IPV4_DF;

```

```

3824     }
3825 }
3826
3827 /*
3828  * If we have an IRE_LOCAL we use the loopback mtu instead of
3829  * the ill for going out the wire i.e., IRE_LOCAL gets the same
3830  * mtu as IRE_LOOPBACK.
3831  */
3832 if (ire->ire_type & (IRE_LOCAL|IRE_LOOPBACK)) {
3833     uint_t loopback_mtu;
3834
3835     loopback_mtu = (ire->ire_ipversion == IPV6_VERSION) ?
3836         ip_loopback_mtu_v6plus : ip_loopback_mtuplus;
3837
3838     if (loopback_mtu < pmtu)
3839         pmtu = loopback_mtu;
3840 } else if (nce != NULL) {
3841     /*
3842      * Make sure we don't exceed the interface MTU.
3843      * In the case of RTF_REJECT or RTF_BLACKHOLE we might not have
3844      * an ill. We'd use the above IP_MAXPACKET in that case just
3845      * to tell the transport something larger than zero.
3846      */
3847     if (ire->ire_type & (IRE_MULTICAST|IRE_BROADCAST)) {
3848         if (nce->ncc_common->ncc_ill->ill_mc_mtu < pmtu)
3849             pmtu = nce->ncc_common->ncc_ill->ill_mc_mtu;
3850         if (nce->ncc_common->ncc_ill != nce->ncc_ill &&
3851             nce->ncc_ill->ill_mc_mtu < pmtu) {
3852             /*
3853              * for interfaces in an IPMP group, the mtu of
3854              * the ncc_ill (under_ill) could be different
3855              * from the mtu of the ncc_ill, so we take the
3856              * min of the two.
3857              */
3858             pmtu = nce->ncc_ill->ill_mc_mtu;
3859         }
3860     } else {
3861         if (nce->ncc_common->ncc_ill->ill_mtu < pmtu)
3862             pmtu = nce->ncc_common->ncc_ill->ill_mtu;
3863         if (nce->ncc_common->ncc_ill != nce->ncc_ill &&
3864             nce->ncc_ill->ill_mtu < pmtu) {
3865             /*
3866              * for interfaces in an IPMP group, the mtu of
3867              * the ncc_ill (under_ill) could be different
3868              * from the mtu of the ncc_ill, so we take the
3869              * min of the two.
3870              */
3871             pmtu = nce->ncc_ill->ill_mtu;
3872         }
3873     }
3874 }
3875
3876 /*
3877  * Handle the IPV6_USE_MIN_MTU socket option or ancillary data.
3878  * Only applies to IPv6.
3879  */
3880 if (!(ixa->ixa_flags & IXAF_IS_IPV4)) {
3881     if (ixa->ixa_flags & IXAF_USE_MIN_MTU) {
3882         switch (ixa->ixa_use_min_mtu) {
3883             case IPV6_USE_MIN_MTU_MULTICAST:
3884                 if (ire->ire_type & IRE_MULTICAST)
3885                     pmtu = IPV6_MIN_MTU;
3886                 break;
3887             case IPV6_USE_MIN_MTU_ALWAYS:
3888                 pmtu = IPV6_MIN_MTU;
3889                 break;

```

```

3890         case IPV6_USE_MIN_MTU_NEVER:
3891             break;
3892     } else {
3893         /* Default is IPV6_USE_MIN_MTU_MULTICAST */
3894         if (ire->ire_type & IRE_MULTICAST)
3895             pmtu = IPV6_MIN_MTU;
3896     }
3897 }
3898
3899 /*
3900  * After receiving an ICMPv6 "packet too big" message with a
3901  * MTU < 1280, and for multirouted IPv6 packets, the IP layer
3902  * will insert a 8-byte fragment header in every packet. We compensate
3903  * for those cases by returning a smaller path MTU to the ULP.
3904  *
3905  * In the case of CGTP then ip_output will add a fragment header.
3906  * Make sure there is room for it by telling a smaller number
3907  * to the transport.
3908  *
3909  * When IXAF_IPV6_ADDR_FRAGHDR we subtract the frag_hdr here
3910  * so the ULPs consistently see a iulp_pmtu and ip_get_pmtu()
3911  * which is the size of the packets it can send.
3912  */
3913 if (!(ixa->ixa_flags & IXAF_IS_IPV4)) {
3914     if ((dce->dce_flags & DCEF_TOO_SMALL_PMTU) ||
3915         (ire->ire_flags & RTF_MULTIRT) ||
3916         (ixa->ixa_flags & IXAF_MULTIRT_MULTICAST)) {
3917         pmtu -= sizeof(ip6_frag_t);
3918         ixa->ixa_flags |= IXAF_IPV6_ADDR_FRAGHDR;
3919     }
3920 }
3921
3922 return (pmtu);
3923 }
3924
3925 /*
3926  * Carve "len" bytes out of an mblk chain, consuming any we empty, and duping
3927  * the final piece where we don't. Return a pointer to the first mblk in the
3928  * result, and update the pointer to the next mblk to chew on. If anything
3929  * goes wrong (i.e., dupb fails), we waste everything in sight and return a
3930  * NULL pointer.
3931  */
3932 mblk_t *
3933 ip_carve_mp(mblk_t **mpp, ssize_t len)
3934 {
3935     mblk_t *mp0;
3936     mblk_t *mpl;
3937     mblk_t *mp2;
3938
3939     if (!len || !mpp || !(mp0 = *mpp))
3940         return (NULL);
3941     /* If we aren't going to consume the first mblk, we need a dup. */
3942     if (mp0->b_wptr - mp0->b_rptr > len) {
3943         mp1 = dupb(mp0);
3944         if (mp1) {
3945             /* Partition the data between the two mblks. */
3946             mp1->b_wptr = mp1->b_rptr + len;
3947             mp0->b_rptr = mp1->b_wptr;
3948             /*
3949              * after adjustments if mblk not consumed is now
3950              * unaligned, try to align it. If this fails free
3951              * all messages and let upper layer recover.
3952              */
3953             if (!OK_32PTR(mp0->b_rptr)) {
3954                 if (!pullupmsg(mp0, -1)) {

```

```

3956         freemsg(mp0);
3957         freemsg(mp1);
3958         *mpp = NULL;
3959         return (NULL);
3960     }
3961 }
3962 }
3963 return (mp1);
3964 }
3965 /* Eat through as many mblks as we need to get len bytes. */
3966 len -= mp0->b_wptr - mp0->b_rptr;
3967 for (mp2 = mp1 = mp0; (mp2 = mp2->b_cont) != 0 && len; mp1 = mp2) {
3968     if (mp2->b_wptr - mp2->b_rptr > len) {
3969         /*
3970          * We won't consume the entire last mblk. Like
3971          * above, dup and partition it.
3972          */
3973         mp1->b_cont = dupb(mp2);
3974         mp1 = mp1->b_cont;
3975         if (!mp1) {
3976             /*
3977              * Trouble. Rather than go to a lot of
3978              * trouble to clean up, we free the messages.
3979              * This won't be any worse than losing it on
3980              * the wire.
3981              */
3982             freemsg(mp0);
3983             freemsg(mp2);
3984             *mpp = NULL;
3985             return (NULL);
3986         }
3987         mp1->b_wptr = mp1->b_rptr + len;
3988         mp2->b_rptr = mp1->b_wptr;
3989         /*
3990          * after adjustments if mblk not consumed is now
3991          * unaligned, try to align it. If this fails free
3992          * all messages and let upper layer recover.
3993          */
3994         if (!OK_32PTR(mp2->b_rptr)) {
3995             if (!pullupmsg(mp2, -1)) {
3996                 freemsg(mp0);
3997                 freemsg(mp2);
3998                 *mpp = NULL;
3999                 return (NULL);
4000             }
4001         }
4002         *mpp = mp2;
4003         return (mp0);
4004     }
4005     /* Decrement len by the amount we just got. */
4006     len -= mp2->b_wptr - mp2->b_rptr;
4007 }
4008 /*
4009  * len should be reduced to zero now. If not our caller has
4010  * screwed up.
4011  */
4012 if (len) {
4013     /* Shouldn't happen! */
4014     freemsg(mp0);
4015     *mpp = NULL;
4016     return (NULL);
4017 }
4018 /*
4019  * We consumed up to exactly the end of an mblk. Detach the part
4020  * we are returning from the rest of the chain.
4021  */

```

```

4022     mp1->b_cont = NULL;
4023     *mpp = mp2;
4024     return (mp0);
4025 }

4027 /* The ill stream is being unplumbed. Called from ip_close */
4028 int
4029 ip_modclose(ill_t *ill)
4030 {
4031     boolean_t success;
4032     ipsq_t *ipsq;
4033     ipif_t *ipif;
4034     queue_t *q = ill->ill_rq;
4035     ip_stack_t *ipst = ill->ill_ipst;
4036     int i;
4037     arl_ill_common_t *ai = ill->ill_common;

4039     /*
4040      * The punlink prior to this may have initiated a capability
4041      * negotiation. But ipsq_enter will block until that finishes or
4042      * times out.
4043      */
4044     success = ipsq_enter(ill, B_FALSE, NEW_OP);

4046     /*
4047      * Open/close/push/pop is guaranteed to be single threaded
4048      * per stream by STREAMS. FS guarantees that all references
4049      * from top are gone before close is called. So there can't
4050      * be another close thread that has set CONDEMNED on this ill.
4051      * and cause ipsq_enter to return failure.
4052      */
4053     ASSERT(success);
4054     ipsq = ill->ill_phyint->phyint_ipsq;

4056     /*
4057      * Mark it condemned. No new reference will be made to this ill.
4058      * Lookup functions will return an error. Threads that try to
4059      * increment the refcnt must check for ILL_CAN_LOOKUP. This ensures
4060      * that the refcnt will drop down to zero.
4061      */
4062     mutex_enter(&ill->ill_lock);
4063     ill->ill_state_flags |= ILL_CONDEMNED;
4064     for (ipif = ill->ill_ipif; ipif != NULL;
4065          ipif = ipif->ipif_next) {
4066         ipif->ipif_state_flags |= IPIF_CONDEMNED;
4067     }
4068     /*
4069      * Wake up anybody waiting to enter the ipsq. ipsq_enter
4070      * returns error if ILL_CONDEMNED is set
4071      */
4072     cv_broadcast(&ill->ill_cv);
4073     mutex_exit(&ill->ill_lock);

4075     /*
4076      * Send all the deferred DLPI messages downstream which came in
4077      * during the small window right before ipsq_enter(). We do this
4078      * without waiting for the ACKs because all the ACKs for M_PROTO
4079      * messages are ignored in ip_rput() when ILL_CONDEMNED is set.
4080      */
4081     ill_dlpi_send_deferred(ill);

4083     /*
4084      * Shut down fragmentation reassembly.
4085      * ill_frag_timer won't start a timer again.
4086      * Now cancel any existing timer
4087      */

```



```

4088 (void) untimeout(ill->ill_frag_timer_id);
4089 (void) ill_frag_timeout(ill, 0);

4091 /*
4092  * Call ill_delete to bring down the ipifs, ilms and ill on
4093  * this ill. Then wait for the refcnts to drop to zero.
4094  * ill_is_freeable checks whether the ill is really quiescent.
4095  * Then make sure that threads that are waiting to enter the
4096  * ipsq have seen the error returned by ipsq_enter and have
4097  * gone away. Then we call ill_delete_tail which does the
4098  * DL_UNBIND_REQ with the driver and then qprocsoff.
4099  */
4100 ill_delete(ill);
4101 mutex_enter(&ill->ill_lock);
4102 while (!ill_is_freeable(ill))
4103     cv_wait(&ill->ill_cv, &ill->ill_lock);

4105 while (ill->ill_waiters)
4106     cv_wait(&ill->ill_cv, &ill->ill_lock);

4108 mutex_exit(&ill->ill_lock);

4110 /*
4111  * ill_delete_tail drops reference on ill_ipst, but we need to keep
4112  * it held until the end of the function since the cleanup
4113  * below needs to be able to use the ip_stack_t.
4114  */
4115 netstack_hold(ipst->ips_netstack);

4117 /* qprocsoff is done via ill_delete_tail */
4118 ill_delete_tail(ill);
4119 /*
4120  * synchronously wait for arp stream to unbind. After this, we
4121  * cannot get any data packets up from the driver.
4122  */
4123 arp_unbind_complete(ill);
4124 ASSERT(ill->ill_ipst == NULL);

4126 /*
4127  * Walk through all conns and qenable those that have queued data.
4128  * Close synchronization needs this to
4129  * be done to ensure that all upper layers blocked
4130  * due to flow control to the closing device
4131  * get unblocked.
4132  */
4133 ipldbg(("ip_wsrv: walking\n"));
4134 for (i = 0; i < TX_FANOUT_SIZE; i++) {
4135     conn_walk_drain(ipst, &ipst->ips_idl_tx_list[i]);
4136 }

4138 /*
4139  * ai can be null if this is an IPv6 ill, or if the IPv4
4140  * stream is being torn down before ARP was plumbed (e.g.,
4141  * /sbin/ifconfig plumbing a stream twice, and encountering
4142  * an error
4143  */
4144 if (ai != NULL) {
4145     ASSERT(!ill->ill_isv6);
4146     mutex_enter(&ai->ai_lock);
4147     ai->ai_ill = NULL;
4148     if (ai->ai_arl == NULL) {
4149         mutex_destroy(&ai->ai_lock);
4150         kmem_free(ai, sizeof (*ai));
4151     } else {
4152         cv_signal(&ai->ai_ill_unplumb_done);
4153         mutex_exit(&ai->ai_lock);

```

```

4154     }
4155 }

4157 mutex_enter(&ipst->ips_ip_mi_lock);
4158 mi_close_unlink(&ipst->ips_ip_g_head, (IDP)ill);
4159 mutex_exit(&ipst->ips_ip_mi_lock);

4161 /*
4162  * credp could be null if the open didn't succeed and ip_modopen
4163  * itself calls ip_close.
4164  */
4165 if (ill->ill_credp != NULL)
4166     crfree(ill->ill_credp);

4168 mutex_destroy(&ill->ill_saved_ire_lock);
4169 mutex_destroy(&ill->ill_lock);
4170 rw_destroy(&ill->ill_mcast_lock);
4171 mutex_destroy(&ill->ill_mcast_serializer);
4172 list_destroy(&ill->ill_nce);

4174 /*
4175  * Now we are done with the module close pieces that
4176  * need the netstack_t.
4177  */
4178 netstack_rele(ipst->ips_netstack);

4180 mi_close_free((IDP)ill);
4181 q->q_ptr = WR(q)->q_ptr = NULL;

4183 ipsq_exit(ipsq);

4185 return (0);
4186 }

4188 /*
4189  * This is called as part of close() for IP, UDP, ICMP, and RTS
4190  * in order to quiesce the conn.
4191  */
4192 void
4193 ip_quiesce_conn(conn_t *connp)
4194 {
4195     boolean_t    drain_cleanup_reqd = B_FALSE;
4196     boolean_t    conn_ioctl_cleanup_reqd = B_FALSE;
4197     boolean_t    ilg_cleanup_reqd = B_FALSE;
4198     ip_stack_t   *ipst;

4200     ASSERT(!IPCL_IS_TCP(connp));
4201     ipst = connp->conn_netstack->netstack_ip;

4203 /*
4204  * Mark the conn as closing, and this conn must not be
4205  * inserted in future into any list. Eg. conn_drain_insert(),
4206  * won't insert this conn into the conn_drain_list.
4207  *
4208  * conn_idl, and conn_ilg cannot get set henceforth.
4209  */
4210 mutex_enter(&connp->conn_lock);
4211 ASSERT(!(connp->conn_state_flags & CONN_QUIESCED));
4212 connp->conn_state_flags |= CONN_CLOSING;
4213 if (connp->conn_idl != NULL)
4214     drain_cleanup_reqd = B_TRUE;
4215 if (connp->conn_oper_pending_ill != NULL)
4216     conn_ioctl_cleanup_reqd = B_TRUE;
4217 if (connp->conn_dhcpinit_ill != NULL) {
4218     ASSERT(connp->conn_dhcpinit_ill->ill_dhcpinit != 0);
4219     atomic_dec_32(&connp->conn_dhcpinit_ill->ill_dhcpinit);

```

```

4220         ill_set_inputfn(connp->conn_dhcpinit_ill);
4221         connp->conn_dhcpinit_ill = NULL;
4222     }
4223     if (connp->conn_ilg != NULL)
4224         ilg_cleanup_reqd = B_TRUE;
4225     mutex_exit(&connp->conn_lock);

4227     if (connp->conn_ioctl_cleanup_reqd)
4228         conn_ioctl_cleanup(connp);

4230     if (is_system_labeled() && connp->conn_anon_port) {
4231         (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
4232             connp->conn_mlp_type, connp->conn_proto,
4233             ntohs(connp->conn_lport), B_FALSE);
4234         connp->conn_anon_port = 0;
4235     }
4236     connp->conn_mlp_type = mlptSingle;

4238     /*
4239      * Remove this conn from any fanout list it is on.
4240      * and then wait for any threads currently operating
4241      * on this endpoint to finish
4242      */
4243     ipcl_hash_remove(connp);

4245     /*
4246      * Remove this conn from the drain list, and do any other cleanup that
4247      * may be required. (TCP conns are never flow controlled, and
4248      * conn_idl will be NULL.)
4249      */
4250     if (drain_cleanup_reqd && connp->conn_idl != NULL) {
4251         idl_t *idl = connp->conn_idl;

4253         mutex_enter(&idl->idl_lock);
4254         conn_drain(connp, B_TRUE);
4255         mutex_exit(&idl->idl_lock);
4256     }

4258     if (connp == ipst->ips_ip_g_mrout)
4259         (void) ip_mrout_done(ipst);

4261     if (ilg_cleanup_reqd)
4262         ilg_delete_all(connp);

4264     /*
4265      * Now conn refcnt can increase only thru CONN_INC_REF_LOCKED.
4266      * callers from write side can't be there now because close
4267      * is in progress. The only other caller is ipcl_walk
4268      * which checks for the condemned flag.
4269      */
4270     mutex_enter(&connp->conn_lock);
4271     connp->conn_state_flags |= CONN_CONDEMNED;
4272     while (connp->conn_ref != 1)
4273         cv_wait(&connp->conn_cv, &connp->conn_lock);
4274     connp->conn_state_flags |= CONN_QUIESCED;
4275     mutex_exit(&connp->conn_lock);
4276 }

4278 /* ARGSUSED */
4279 int
4280 ip_close(queue_t *q, int flags)
4281 {
4282     conn_t      *connp;

4284     /*
4285      * Call the appropriate delete routine depending on whether this is

```

```

4286     * a module or device.
4287     */
4288     if (WR(q)->q_next != NULL) {
4289         /* This is a module close */
4290         return (ip_modclose((ill_t *)q->q_ptr));
4291     }

4293     connp = q->q_ptr;
4294     ip_quiesce_conn(connp);

4296     qprocsoff(q);

4298     /*
4299      * Now we are truly single threaded on this stream, and can
4300      * delete the things hanging off the connp, and finally the connp.
4301      * We removed this connp from the fanout list, it cannot be
4302      * accessed thru the fanouts, and we already waited for the
4303      * conn_ref to drop to 0. We are already in close, so
4304      * there cannot be any other thread from the top. qprocsoff
4305      * has completed, and service has completed or won't run in
4306      * future.
4307      */
4308     ASSERT(connp->conn_ref == 1);

4310     inet_minor_free(connp->conn_minor_arena, connp->conn_dev);

4312     connp->conn_ref--;
4313     ipcl_conn_destroy(connp);

4315     q->q_ptr = WR(q)->q_ptr = NULL;
4316     return (0);
4317 }

4319 /*
4320  * Wrapper around putnext() so that ip_rts_request can merely use
4321  * conn_recv.
4322  */
4323 /*ARGSUSED2*/
4324 static void
4325 ip_conn_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
4326 {
4327     conn_t *connp = (conn_t *)arg1;

4329     putnext(connp->conn_rq, mp);
4330 }

4332 /* Dummy in case ICMP error delivery is attempted to a /dev/ip instance */
4333 /* ARGSUSED */
4334 static void
4335 ip_conn_input_icmp(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
4336 {
4337     freemsg(mp);
4338 }

4340 /*
4341  * Called when the module is about to be unloaded
4342  */
4343 void
4344 ip_ddi_destroy(void)
4345 {
4346     /* This needs to be called before destroying any transports. */
4347     mutex_enter(&cpu_lock);
4348     unregister_cpu_setup_func(ip_tp_cpu_update, NULL);
4349     mutex_exit(&cpu_lock);

4351     tnet_fini();

```

```

4353     icmp_ddi_g_destroy();
4354     rts_ddi_g_destroy();
4355     udp_ddi_g_destroy();
4356     dccp_ddi_g_destroy();
4357 #endif /* ! codereview */
4358     sctp_ddi_g_destroy();
4359     tcp_ddi_g_destroy();
4360     ilb_ddi_g_destroy();
4361     dce_g_destroy();
4362     ipsec_policy_g_destroy();
4363     ipcl_g_destroy();
4364     ip_net_g_destroy();
4365     ip_ire_g_fini();
4366     inet_minor_destroy(ip_minor_arena_sa);
4367 #if defined(_LP64)
4368     inet_minor_destroy(ip_minor_arena_la);
4369 #endif

4371 #ifdef DEBUG
4372     list_destroy(&ip_thread_list);
4373     rw_destroy(&ip_thread_rwlock);
4374     tsd_destroy(&ip_thread_data);
4375 #endif

4377     netstack_unregister(NS_IP);
4378 }

4380 /*
4381  * First step in cleanup.
4382  */
4383 /* ARGSUSED */
4384 static void
4385 ip_stack_shutdown(netstackid_t stackid, void *arg)
4386 {
4387     ip_stack_t *ipst = (ip_stack_t *)arg;

4389 #ifdef NS_DEBUG
4390     printf("ip_stack_shutdown(%p, stack %d)\n", (void *)ipst, stackid);
4391 #endif

4393     /*
4394      * Perform cleanup for special interfaces (loopback and IPMP).
4395      */
4396     ip_interface_cleanup(ipst);

4398     /*
4399      * The *_hook_shutdown()s start the process of notifying any
4400      * consumers that things are going away.... nothing is destroyed.
4401      */
4402     ipv4_hook_shutdown(ipst);
4403     ipv6_hook_shutdown(ipst);
4404     arp_hook_shutdown(ipst);

4406     mutex_enter(&ipst->ips_capab_taskq_lock);
4407     ipst->ips_capab_taskq_quit = B_TRUE;
4408     cv_signal(&ipst->ips_capab_taskq_cv);
4409     mutex_exit(&ipst->ips_capab_taskq_lock);
4410 }

4412 /*
4413  * Free the IP stack instance.
4414  */
4415 static void
4416 ip_stack_fini(netstackid_t stackid, void *arg)
4417 {

```

```

4418     ip_stack_t *ipst = (ip_stack_t *)arg;
4419     int ret;

4421 #ifdef NS_DEBUG
4422     printf("ip_stack_fini(%p, stack %d)\n", (void *)ipst, stackid);
4423 #endif

4424     /*
4425      * At this point, all of the notifications that the events and
4426      * protocols are going away have been run, meaning that we can
4427      * now set about starting to clean things up.
4428      */
4429     ipobs_fini(ipst);
4430     ipv4_hook_destroy(ipst);
4431     ipv6_hook_destroy(ipst);
4432     arp_hook_destroy(ipst);
4433     ip_net_destroy(ipst);

4435     ipmp_destroy(ipst);

4437     ip_kstat_fini(stackid, ipst->ips_ip_mibkp);
4438     ipst->ips_ip_mibkp = NULL;
4439     icmp_kstat_fini(stackid, ipst->ips_icmp_mibkp);
4440     ipst->ips_icmp_mibkp = NULL;
4441     ip_kstat2_fini(stackid, ipst->ips_ip_kstat);
4442     ipst->ips_ip_kstat = NULL;
4443     bzero(&ipst->ips_ip_statistics, sizeof (ipst->ips_ip_statistics));
4444     ip6_kstat_fini(stackid, ipst->ips_ip6_kstat);
4445     ipst->ips_ip6_kstat = NULL;
4446     bzero(&ipst->ips_ip6_statistics, sizeof (ipst->ips_ip6_statistics));

4448     kmem_free(ipst->ips_propinfo_tbl,
4449               ip_propinfo_count * sizeof (mod_prop_info_t));
4450     ipst->ips_propinfo_tbl = NULL;

4452     dce_stack_destroy(ipst);
4453     ip_mrouter_stack_destroy(ipst);

4455     ret = untimeout(ipst->ips_igmp_timeout_id);
4456     if (ret == -1) {
4457         ASSERT(ipst->ips_igmp_timeout_id == 0);
4458     } else {
4459         ASSERT(ipst->ips_igmp_timeout_id != 0);
4460         ipst->ips_igmp_timeout_id = 0;
4461     }
4462     ret = untimeout(ipst->ips_igmp_slowtimeout_id);
4463     if (ret == -1) {
4464         ASSERT(ipst->ips_igmp_slowtimeout_id == 0);
4465     } else {
4466         ASSERT(ipst->ips_igmp_slowtimeout_id != 0);
4467         ipst->ips_igmp_slowtimeout_id = 0;
4468     }
4469     ret = untimeout(ipst->ips_mld_timeout_id);
4470     if (ret == -1) {
4471         ASSERT(ipst->ips_mld_timeout_id == 0);
4472     } else {
4473         ASSERT(ipst->ips_mld_timeout_id != 0);
4474         ipst->ips_mld_timeout_id = 0;
4475     }
4476     ret = untimeout(ipst->ips_mld_slowtimeout_id);
4477     if (ret == -1) {
4478         ASSERT(ipst->ips_mld_slowtimeout_id == 0);
4479     } else {
4480         ASSERT(ipst->ips_mld_slowtimeout_id != 0);
4481         ipst->ips_mld_slowtimeout_id = 0;
4482     }

```

```

4484     ip_ire_fini(ipst);
4485     ip6_asp_free(ipst);
4486     conn_drain_fini(ipst);
4487     ipcl_destroy(ipst);

4489     mutex_destroy(&ipst->ips_ndp4->ndp_g_lock);
4490     mutex_destroy(&ipst->ips_ndp6->ndp_g_lock);
4491     kmem_free(ipst->ips_ndp4, sizeof (ndp_g_t));
4492     ipst->ips_ndp4 = NULL;
4493     kmem_free(ipst->ips_ndp6, sizeof (ndp_g_t));
4494     ipst->ips_ndp6 = NULL;

4496     if (ipst->ips_loopback_ksp != NULL) {
4497         kstat_delete_netstack(ipst->ips_loopback_ksp, stackid);
4498         ipst->ips_loopback_ksp = NULL;
4499     }

4501     mutex_destroy(&ipst->ips_capab_taskq_lock);
4502     cv_destroy(&ipst->ips_capab_taskq_cv);

4504     rw_destroy(&ipst->ips_srcid_lock);

4506     mutex_destroy(&ipst->ips_ip_mi_lock);
4507     rw_destroy(&ipst->ips_ill_g_usesrc_lock);

4509     mutex_destroy(&ipst->ips_igmp_timer_lock);
4510     mutex_destroy(&ipst->ips_mld_timer_lock);
4511     mutex_destroy(&ipst->ips_igmp_slowtimeout_lock);
4512     mutex_destroy(&ipst->ips_mld_slowtimeout_lock);
4513     mutex_destroy(&ipst->ips_ip_addr_avail_lock);
4514     rw_destroy(&ipst->ips_ill_g_lock);

4516     kmem_free(ipst->ips_phyint_g_list, sizeof (phyint_list_t));
4517     ipst->ips_phyint_g_list = NULL;
4518     kmem_free(ipst->ips_ill_g_heads, sizeof (ill_g_head_t) * MAX_G_HEADS);
4519     ipst->ips_ill_g_heads = NULL;

4521     ldi_ident_release(ipst->ips_ldi_ident);
4522     kmem_free(ipst, sizeof (*ipst));
4523 }

4525 /*
4526  * This function is called from the TSD destructor, and is used to debug
4527  * reference count issues in IP. See block comment in <inet/ip_if.h> for
4528  * details.
4529  */
4530 static void
4531 ip_thread_exit(void *phash)
4532 {
4533     th_hash_t *thh = phash;

4535     rw_enter(&ip_thread_rwlock, RW_WRITER);
4536     list_remove(&ip_thread_list, thh);
4537     rw_exit(&ip_thread_rwlock);
4538     mod_hash_destroy_hash(thh->thh_hash);
4539     kmem_free(thh, sizeof (*thh));
4540 }

4542 /*
4543  * Called when the IP kernel module is loaded into the kernel
4544  */
4545 void
4546 ip_ddi_init(void)
4547 {
4548     ip_squeue_flag = ip_squeue_switch(ip_squeue_enter);

```

```

4550     /*
4551      * For IP and TCP the minor numbers should start from 2 since we have 4
4552      * initial devices: ip, ip6, tcp, tcp6.
4553      */
4554     /*
4555      * If this is a 64-bit kernel, then create two separate arenas -
4556      * one for TLIs in the range of INET_MIN_DEV+2 through 2^^18-1, and the
4557      * other for socket apps in the range 2^^18 through 2^^32-1.
4558      */
4559     ip_minor_arena_la = NULL;
4560     ip_minor_arena_sa = NULL;
4561     #if defined(_LP64)
4562     if ((ip_minor_arena_sa = inet_minor_create("ip_minor_arena_sa",
4563         INET_MIN_DEV + 2, MAXMIN32, KM_SLEEP)) == NULL) {
4564         cmn_err(CE_PANIC,
4565             "ip_ddi_init: ip_minor_arena_sa creation failed\n");
4566     }
4567     if ((ip_minor_arena_la = inet_minor_create("ip_minor_arena_la",
4568         MAXMIN32 + 1, MAXMIN64, KM_SLEEP)) == NULL) {
4569         cmn_err(CE_PANIC,
4570             "ip_ddi_init: ip_minor_arena_la creation failed\n");
4571     }
4572     #else
4573     if ((ip_minor_arena_sa = inet_minor_create("ip_minor_arena_sa",
4574         INET_MIN_DEV + 2, MAXMIN, KM_SLEEP)) == NULL) {
4575         cmn_err(CE_PANIC,
4576             "ip_ddi_init: ip_minor_arena_sa creation failed\n");
4577     }
4578     #endif
4579     ip_poll_normal_ticks = MSEC_TO_TICK_ROUNDUP(ip_poll_normal_ms);

4581     ipcl_g_init();
4582     ip_ire_g_init();
4583     ip_net_g_init();

4585     #ifdef DEBUG
4586     tsd_create(&ip_thread_data, ip_thread_exit);
4587     rw_init(&ip_thread_rwlock, NULL, RW_DEFAULT, NULL);
4588     list_create(&ip_thread_list, sizeof (th_hash_t),
4589         offsetof(th_hash_t, thh_link));
4590     #endif
4591     ipsec_policy_g_init();
4592     tcp_ddi_g_init();
4593     sctp_ddi_g_init();
4594     dccp_ddi_g_init();
4595     #endif /* ! codereview */
4596     dce_g_init();

4598     /*
4599      * We want to be informed each time a stack is created or
4600      * destroyed in the kernel, so we can maintain the
4601      * set of udp_stack_t's.
4602      */
4603     netstack_register(NS_IP, ip_stack_init, ip_stack_shutdown,
4604         ip_stack_fini);

4606     tnet_init();

4608     udp_ddi_g_init();
4609     rts_ddi_g_init();
4610     icmp_ddi_g_init();
4611     ilb_ddi_g_init();

4613     /* This needs to be called after all transports are initialized. */
4614     mutex_enter(&cpu_lock);
4615     register_cpu_setup_func(ip_tp_cpu_update, NULL);

```

```

4616     mutex_exit(&cpu_lock);
4617 }

4619 /*
4620  * Initialize the IP stack instance.
4621  */
4622 static void *
4623 ip_stack_init(netstackid_t stackid, netstack_t *ns)
4624 {
4625     ip_stack_t      *ipst;
4626     size_t          arrsz;
4627     major_t          major;

4629 #ifdef NS_DEBUG
4630     printf("ip_stack_init(stack %d)\n", stackid);
4631 #endif

4633     ipst = (ip_stack_t *)kmem_zalloc(sizeof(*ipst), KM_SLEEP);
4634     ipst->ips_netstack = ns;

4636     ipst->ips_ill_g_heads = kmem_zalloc(sizeof(ill_g_head_t) * MAX_G_HEADS,
4637     KM_SLEEP);
4638     ipst->ips_phyint_g_list = kmem_zalloc(sizeof(phyint_list_t),
4639     KM_SLEEP);
4640     ipst->ips_ndp4 = kmem_zalloc(sizeof(ndp_g_t), KM_SLEEP);
4641     ipst->ips_ndp6 = kmem_zalloc(sizeof(ndp_g_t), KM_SLEEP);
4642     mutex_init(&ipst->ips_ndp4->ndp_g_lock, NULL, MUTEX_DEFAULT, NULL);
4643     mutex_init(&ipst->ips_ndp6->ndp_g_lock, NULL, MUTEX_DEFAULT, NULL);

4645     mutex_init(&ipst->ips_igmp_timer_lock, NULL, MUTEX_DEFAULT, NULL);
4646     ipst->ips_igmp_deferred_next = INFINITY;
4647     mutex_init(&ipst->ips_mld_timer_lock, NULL, MUTEX_DEFAULT, NULL);
4648     ipst->ips_mld_deferred_next = INFINITY;
4649     mutex_init(&ipst->ips_igmp_slowtimeout_lock, NULL, MUTEX_DEFAULT, NULL);
4650     mutex_init(&ipst->ips_mld_slowtimeout_lock, NULL, MUTEX_DEFAULT, NULL);
4651     mutex_init(&ipst->ips_ip_mi_lock, NULL, MUTEX_DEFAULT, NULL);
4652     mutex_init(&ipst->ips_ip_addr_avail_lock, NULL, MUTEX_DEFAULT, NULL);
4653     rw_init(&ipst->ips_ill_g_lock, NULL, RW_DEFAULT, NULL);
4654     rw_init(&ipst->ips_ill_g_usersrc_lock, NULL, RW_DEFAULT, NULL);

4656     ipcl_init(ipst);
4657     ip_ire_init(ipst);
4658     ip6_asp_init(ipst);
4659     ipif_init(ipst);
4660     conn_drain_init(ipst);
4661     ip_mroutestack_init(ipst);
4662     dce_stack_init(ipst);

4664     ipst->ips_ip_multirt_log_interval = 1000;

4666     ipst->ips_ill_index = 1;

4668     ipst->ips_saved_ip_forwarding = -1;
4669     ipst->ips_req_vif_num = ALL_VIFS; /* Index to Register vif */

4671     arrsz = ip_propinfo_count * sizeof(mod_prop_info_t);
4672     ipst->ips_propinfo_tbl = (mod_prop_info_t *)kmem_alloc(arrsz, KM_SLEEP);
4673     bcopy(ip_propinfo_tbl, ipst->ips_propinfo_tbl, arrsz);

4675     ipst->ips_ip_mibkp = ip_kstat_init(stackid, ipst);
4676     ipst->ips_icmp_mibkp = icmp_kstat_init(stackid);
4677     ipst->ips_ip_kstat = ip_kstat2_init(stackid, &ipst->ips_ip_statistics);
4678     ipst->ips_ip6_kstat =
4679         ip6_kstat_init(stackid, &ipst->ips_ip6_statistics);

4681     ipst->ips_ip_src_id = 1;

```

```

4682     rw_init(&ipst->ips_srcid_lock, NULL, RW_DEFAULT, NULL);

4684     ipst->ips_src_generation = SRC_GENERATION_INITIAL;

4686     ip_net_init(ipst, ns);
4687     ipv4_hook_init(ipst);
4688     ipv6_hook_init(ipst);
4689     arp_hook_init(ipst);
4690     ipmp_init(ipst);
4691     ipobs_init(ipst);

4693     /*
4694      * Create the taskq dispatcher thread and initialize related stuff.
4695      */
4696     mutex_init(&ipst->ips_capab_taskq_lock, NULL, MUTEX_DEFAULT, NULL);
4697     cv_init(&ipst->ips_capab_taskq_cv, NULL, CV_DEFAULT, NULL);
4698     ipst->ips_capab_taskq_thread = thread_create(NULL, 0,
4699         ill_taskq_dispatch, ipst, 0, &p0, TS_RUN, minclsyspri);

4701     major = mod_name_to_major(INET_NAME);
4702     (void) ldi_ident_from_major(major, &ipst->ips_ldi_ident);
4703     return(ipst);
4704 }

4706 /*
4707  * Allocate and initialize a DLPI template of the specified length. (May be
4708  * called as writer.)
4709  */
4710 mblk_t *
4711 ip_dlpi_alloc(size_t len, t_uscalar_t prim)
4712 {
4713     mblk_t *mp;

4715     mp = allocb(len, BPRI_MED);
4716     if (!mp)
4717         return(NULL);

4719     /*
4720      * DLPIv2 says that DL_INFO_REQ and DL_TOKEN_REQ (the latter
4721      * of which we don't seem to use) are sent with M_PCPROTO, and
4722      * that other DLPI are M_PROTO.
4723      */
4724     if (prim == DL_INFO_REQ) {
4725         mp->b_datap->db_type = M_PCPROTO;
4726     } else {
4727         mp->b_datap->db_type = M_PROTO;
4728     }

4730     mp->b_wptr = mp->b_rptr + len;
4731     bzero(mp->b_rptr, len);
4732     ((dl_unitdata_req_t *)mp->b_rptr)->dl_primitive = prim;
4733     return(mp);
4734 }

4736 /*
4737  * Allocate and initialize a DLPI notification. (May be called as writer.)
4738  */
4739 mblk_t *
4740 ip_dlnotify_alloc(uint_t notification, uint_t data)
4741 {
4742     dl_notify_ind_t *notifyp;
4743     mblk_t *mp;

4745     if ((mp = ip_dlpi_alloc(DL_NOTIFY_IND_SIZE, DL_NOTIFY_IND)) == NULL)
4746         return(NULL);

```

```

4748     notifyf = (dl_notify_ind_t *)mp->b_rptr;
4749     notifyf->dl_notification = notification;
4750     notifyf->dl_data = data;
4751     return (mp);
4752 }

4754 mblk_t *
4755 ip_dlnotify_alloc2(uint_t notification, uint_t data1, uint_t data2)
4756 {
4757     dl_notify_ind_t *notifyf;
4758     mblk_t *mp;

4760     if ((mp = ip_dlpi_alloc(DL_NOTIFY_IND_SIZE, DL_NOTIFY_IND)) == NULL)
4761         return (NULL);

4763     notifyf = (dl_notify_ind_t *)mp->b_rptr;
4764     notifyf->dl_notification = notification;
4765     notifyf->dl_data1 = data1;
4766     notifyf->dl_data2 = data2;
4767     return (mp);
4768 }

4770 /*
4771  * Debug formatting routine. Returns a character string representation of the
4772  * addr in buf, of the form xxx.xxx.xxx.xxx. This routine takes the address
4773  * in the form of a ipaddr_t and calls ip_dot_saddr with a pointer.
4774  *
4775  * Once the ndd table-printing interfaces are removed, this can be changed to
4776  * standard dotted-decimal form.
4777  */
4778 char *
4779 ip_dot_addr(ipaddr_t addr, char *buf)
4780 {
4781     uint8_t *ap = (uint8_t *)&addr;

4783     (void) mi_sprintf(buf, "%03d.%03d.%03d.%03d",
4784         ap[0] & 0xFF, ap[1] & 0xFF, ap[2] & 0xFF, ap[3] & 0xFF);
4785     return (buf);
4786 }

4788 /*
4789  * Write the given MAC address as a printable string in the usual colon-
4790  * separated format.
4791  */
4792 const char *
4793 mac_colon_addr(const uint8_t *addr, size_t alen, char *buf, size_t buflen)
4794 {
4795     char *bp;

4797     if (alen == 0 || buflen < 4)
4798         return ("?");
4799     bp = buf;
4800     for (;;) {
4801         /*
4802          * If there are more MAC address bytes available, but we won't
4803          * have any room to print them, then add "." to the string
4804          * instead. See below for the 'magic number' explanation.
4805          */
4806         if ((alen == 2 && buflen < 6) || (alen > 2 && buflen < 7)) {
4807             (void) strcpy(bp, "...");
4808             break;
4809         }
4810         (void) sprintf(bp, "%02x", *addr++);
4811         bp += 2;
4812         if (--alen == 0)
4813             break;

```

```

4814         *bp++ = ':';
4815         buflen -= 3;
4816     }
4817     /*
4818      * At this point, based on the first 'if' statement above,
4819      * either alen == 1 and buflen >= 3, or alen > 1 and
4820      * buflen >= 4. The first case leaves room for the final "xx"
4821      * number and trailing NUL byte. The second leaves room for at
4822      * least "...". Thus the apparently 'magic' numbers chosen for
4823      * that statement.
4824     */
4825     return (buf);
4826 }

4828 /*
4829  * Called when it is conceptually a ULP that would sent the packet
4830  * e.g., port unreachable and protocol unreachable. Check that the packet
4831  * would have passed the IPsec global policy before sending the error.
4832  *
4833  * Send an ICMP error after patching up the packet appropriately.
4834  * Uses ip_drop_input and bumps the appropriate MIB.
4835  */
4836 void
4837 ip_fanout_send_icmp_v4(mblk_t *mp, uint_t icmp_type, uint_t icmp_code,
4838     ip_rcv_attr_t *ira)
4839 {
4840     ipha_t *ipha;
4841     boolean_t secure;
4842     ill_t *ill = ira->ira_ill;
4843     ip_stack_t *ipst = ill->ill_ipst;
4844     netstack_t *ns = ipst->ips_netstack;
4845     ipsec_stack_t *ipss = ns->netstack_ipsec;

4847     secure = ira->ira_flags & IRAF_IPSEC_SECURE;

4849     /*
4850      * We are generating an icmp error for some inbound packet.
4851      * Called from all ip_fanout(udp, tcp, proto) functions.
4852      * Before we generate an error, check with global policy
4853      * to see whether this is allowed to enter the system. As
4854      * there is no "conn", we are checking with global policy.
4855      */
4856     ipha = (ipha_t *)mp->b_rptr;
4857     if (secure || ipss->ipsec_inbound_v4_policy_present) {
4858         mp = ipsec_check_global_policy(mp, NULL, ipha, NULL, ira, ns);
4859         if (mp == NULL)
4860             return;
4861     }

4863     /* We never send errors for protocols that we do implement */
4864     if (ira->ira_protocol == IPPROTO_ICMP ||
4865         ira->ira_protocol == IPPROTO_IGMP) {
4866         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
4867         ip_drop_input("ip_fanout_send_icmp_v4", mp, ill);
4868         freemsg(mp);
4869         return;
4870     }
4871     /*
4872      * Have to correct checksum since
4873      * the packet might have been
4874      * fragmented and the reassembly code in ip_rput
4875      * does not restore the IP checksum.
4876      */
4877     ipha->ipha_hdr_checksum = 0;
4878     ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

```

```

4880     switch (icmp_type) {
4881     case ICMP_DEST_UNREACHABLE:
4882         switch (icmp_code) {
4883         case ICMP_PROTOCOL_UNREACHABLE:
4884             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInUnknownProtos);
4885             ip_drop_input("ipIfStatsInUnknownProtos", mp, ill);
4886             break;
4887         case ICMP_PORT_UNREACHABLE:
4888             BUMP_MIB(ill->ill_ip_mib, udpIfStatsNoPorts);
4889             ip_drop_input("ipIfStatsNoPorts", mp, ill);
4890             break;
4891         }
4892
4893         icmp_unreachable(mp, icmp_code, ira);
4894         break;
4895     default:
4896 #ifdef DEBUG
4897         panic("ip_fanout_send_icmp_v4: wrong type");
4898         /*NOTREACHED*/
4899 #else
4900         freemsg(mp);
4901         break;
4902 #endif
4903     }
4904 }
4905
4906 /*
4907  * Used to send an ICMP error message when a packet is received for
4908  * a protocol that is not supported. The mblk passed as argument
4909  * is consumed by this function.
4910  */
4911 void
4912 ip_proto_not_sup(mblk_t *mp, ip_recv_attr_t *ira)
4913 {
4914     ipha_t      *ipha;
4915
4916     ipha = (ipha_t *)mp->b_rptr;
4917     if (ira->ira_flags & IRAF_IS_IPV4) {
4918         ASSERT(IPH_HDR_VERSION(ipha) == IP_VERSION);
4919         ip_fanout_send_icmp_v4(mp, ICMP_DEST_UNREACHABLE,
4920             ICMP_PROTOCOL_UNREACHABLE, ira);
4921     } else {
4922         ASSERT(IPH_HDR_VERSION(ipha) == IPV6_VERSION);
4923         ip_fanout_send_icmp_v6(mp, ICMP6_PARAM_PROB,
4924             ICMP6_PARAMPROB_NEXTHEADER, ira);
4925     }
4926 }
4927
4928 /*
4929  * Deliver a rawip packet to the given conn, possibly applying ipsec policy.
4930  * Handles IPv4 and IPv6.
4931  * We are responsible for disposing of mp, such as by freemsg() or putnext()
4932  * Caller is responsible for dropping references to the conn.
4933  */
4934 void
4935 ip_fanout_proto_conn(conn_t *connp, mblk_t *mp, ipha_t *ipha, ip6_t *ip6h,
4936     ip_recv_attr_t *ira)
4937 {
4938     ill_t      *ill = ira->ira_ill;
4939     ip_stack_t *ipst = ill->ill_ipst;
4940     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;
4941     boolean_t   secure;
4942     uint_t      protocol = ira->ira_protocol;
4943     iraflags_t  iraflags = ira->ira_flags;
4944     queue_t     *rq;

```

```

4946     secure = iraflags & IRAF_IPSEC_SECURE;
4947
4948     rq = connp->conn_rq;
4949     if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrlid : !canputnext(rq)) {
4950         switch (protocol) {
4951         case IPPROTO_ICMPV6:
4952             BUMP_MIB(ill->ill_icmp6_mib, ipv6IfIcmpInOverflows);
4953             break;
4954         case IPPROTO_ICMP:
4955             BUMP_MIB(&ipst->ips_icmp_mib, icmpInOverflows);
4956             break;
4957         default:
4958             BUMP_MIB(ill->ill_ip_mib, rawipIfStatsInOverflows);
4959             break;
4960         }
4961         freemsg(mp);
4962         return;
4963     }
4964
4965     ASSERT(!(IPCL_IS_IPTUN(connp)));
4966
4967     if (((iraflags & IRAF_IS_IPV4) ?
4968         CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
4969         CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
4970         secure) {
4971         mp = ipsec_check_inbound_policy(mp, connp, ipha,
4972             ip6h, ira);
4973         if (mp == NULL) {
4974             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
4975             /* Note that mp is NULL */
4976             ip_drop_input("ipIfStatsInDiscards", mp, ill);
4977             return;
4978         }
4979     }
4980
4981     if (iraflags & IRAF_ICMP_ERROR) {
4982         (connp->conn_recvicmp)(connp, mp, NULL, ira);
4983     } else {
4984         ill_t *rill = ira->ira_rill;
4985
4986         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
4987         ira->ira_ill = ira->ira_rill = NULL;
4988         /* Send it upstream */
4989         (connp->conn_rcv)(connp, mp, NULL, ira);
4990         ira->ira_ill = ill;
4991         ira->ira_rill = rill;
4992     }
4993 }
4994
4995 /*
4996  * Handle protocols with which IP is less intimate. There
4997  * can be more than one stream bound to a particular
4998  * protocol. When this is the case, normally each one gets a copy
4999  * of any incoming packets.
5000  *
5001  * IPsec NOTE :
5002  *
5003  * Don't allow a secure packet going up a non-secure connection.
5004  * We don't allow this because
5005  *
5006  * 1) Reply might go out in clear which will be dropped at
5007  *    the sending side.
5008  * 2) If the reply goes out in clear it will give the
5009  *    adversary enough information for getting the key in
5010  *    most of the cases.
5011  */

```

```

5012 * Moreover getting a secure packet when we expect clear
5013 * implies that SA's were added without checking for
5014 * policy on both ends. This should not happen once ISAKMP
5015 * is used to negotiate SAs as SAs will be added only after
5016 * verifying the policy.
5017 */
5018 * Zones notes:
5019 * Earlier in ip_input on a system with multiple shared-IP zones we
5020 * duplicate the multicast and broadcast packets and send them up
5021 * with each explicit zoneid that exists on that ill.
5022 * This means that here we can match the zoneid with SO_ALLZONES being special.
5023 */
5024 void
5025 ip_fanout_proto_v4(mblk_t *mp, ipha_t *ipha, ip_rcv_attr_t *ira)
5026 {
5027     mblk_t      *mpl;
5028     ipaddr_t     laddr;
5029     conn_t       *connp, *first_connp, *next_connp;
5030     connf_t      *connfp;
5031     ill_t        *ill = ira->ira_ill;
5032     ip_stack_t   *ipst = ill->ill_ipst;

5034     laddr = ipha->ipha_dst;

5036     connfp = &ipst->ips_ipcl_proto_fanout_v4[ira->ira_protocol];
5037     mutex_enter(&connfp->connf_lock);
5038     connp = connfp->connf_head;
5039     for (connp = connfp->connf_head; connp != NULL;
5040         connp = connp->conn_next) {
5041         /* Note: IPCL_PROTO_MATCH includes conn_wantpacket */
5042         if (IPCL_PROTO_MATCH(connp, ira, ipha) &&
5043             (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5044             tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp))) {
5045             break;
5046         }
5047     }

5049     if (connp == NULL) {
5050         /*
5051          * No one bound to these addresses. Is
5052          * there a client that wants all
5053          * unclaimed datagrams?
5054          */
5055         mutex_exit(&connfp->connf_lock);
5056         ip_fanout_send_icmp_v4(mp, ICMP_DEST_UNREACHABLE,
5057             ICMP_PROTOCOL_UNREACHABLE, ira);
5058         return;
5059     }

5061     ASSERT(IPCL_IS_NONSTR(connp) || connp->conn_rq != NULL);

5063     CONN_INC_REF(connp);
5064     first_connp = connp;
5065     connp = connp->conn_next;

5067     for (;;) {
5068         while (connp != NULL) {
5069             /* Note: IPCL_PROTO_MATCH includes conn_wantpacket */
5070             if (IPCL_PROTO_MATCH(connp, ira, ipha) &&
5071                 (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5072                 tsol_receive_local(mp, &laddr, IPV4_VERSION,
5073                     ira, connp)))
5074                 break;
5075             connp = connp->conn_next;
5076         }

```

```

5078         if (connp == NULL) {
5079             /* No more interested clients */
5080             connp = first_connp;
5081             break;
5082         }
5083         if (((mpl = dupmsg(mp)) == NULL) &&
5084             ((mpl = copymsg(mp)) == NULL)) {
5085             /* Memory allocation failed */
5086             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5087             ip_drop_input("ipIfStatsInDiscards", mp, ill);
5088             connp = first_connp;
5089             break;
5090         }

5092         CONN_INC_REF(connp);
5093         mutex_exit(&connfp->connf_lock);

5095         ip_fanout_proto_conn(connp, mpl, (ipha_t *)mpl->b_rptr, NULL,
5096             ira);

5098         mutex_enter(&connfp->connf_lock);
5099         /* Follow the next pointer before releasing the conn. */
5100         next_connp = connp->conn_next;
5101         CONN_DEC_REF(connp);
5102         connp = next_connp;
5103     }

5105     /* Last one. Send it upstream. */
5106     mutex_exit(&connfp->connf_lock);

5108     ip_fanout_proto_conn(connp, mp, ipha, NULL, ira);

5110     CONN_DEC_REF(connp);
5111 }

5113 /*
5114 * If we have a IPsec NAT-Traversal packet, strip the zero-SPI or
5115 * pass it along to ESP if the SPI is non-zero. Returns the mblk if the mblk
5116 * is not consumed.
5117 *
5118 * One of three things can happen, all of which affect the passed-in mblk:
5119 *
5120 * 1.) The packet is stock UDP and gets its zero-SPI stripped. Return mblk..
5121 *
5122 * 2.) The packet is ESP-in-UDP, gets transformed into an equivalent
5123 *     ESP packet, and is passed along to ESP for consumption. Return NULL.
5124 *
5125 * 3.) The packet is an ESP-in-UDP Keepalive. Drop it and return NULL.
5126 */
5127 mblk_t *
5128 zero_spi_check(mblk_t *mp, ip_rcv_attr_t *ira)
5129 {
5130     int shift, plen, iph_len;
5131     ipha_t *ipha;
5132     udpha_t *udpha;
5133     uint32_t *spi;
5134     uint32_t esp_ports;
5135     uint8_t *orptra;
5136     ip_stack_t *ipst = ira->ira_ill->ill_ipst;
5137     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;

5139     ipha = (ipha_t *)mp->b_rptr;
5140     iph_len = ira->ira_ip_hdr_length;
5141     plen = ira->ira_pktlen;

5143     if (plen - iph_len - sizeof(udpha_t) < sizeof(uint32_t)) {

```



```

5144      /*
5145       * Most likely a keepalive for the benefit of an intervening
5146       * NAT. These aren't for us, per se, so drop it.
5147       */
5148       * RFC 3947/8 doesn't say for sure what to do for 2-3
5149       * byte packets (keepalives are 1-byte), but we'll drop them
5150       * also.
5151       */
5152       ip_drop_packet(mp, B_TRUE, ira->ira_ill,
5153         DROPPER(ipss, ipds_esp_nat_t_ka), &ipss->ipsec_dropper);
5154       return (NULL);
5155     }

5157     if (MBLKL(mp) < iph_len + sizeof (udpha_t) + sizeof (*spi)) {
5158       /* might as well pull it all up - it might be ESP. */
5159       if (!pullupmsg(mp, -1)) {
5160         ip_drop_packet(mp, B_TRUE, ira->ira_ill,
5161           DROPPER(ipss, ipds_esp_nomem),
5162           &ipss->ipsec_dropper);
5163         return (NULL);
5164       }
5165     }

5166     ipha = (ipha_t *)mp->b_rptr;
5167   }
5168   spi = (uint32_t *) (mp->b_rptr + iph_len + sizeof (udpha_t));
5169   if (*spi == 0) {
5170     /* UDP packet - remove 0-spi. */
5171     shift = sizeof (uint32_t);
5172   } else {
5173     /* ESP-in-UDP packet - reduce to ESP. */
5174     ipha->ipha_protocol = IPPROTO_ESP;
5175     shift = sizeof (udpha_t);
5176   }

5178   /* Fix IP header */
5179   ira->ira_pktlen = (plen - shift);
5180   ipha->ipha_length = htons(ira->ira_pktlen);
5181   ipha->ipha_hdr_checksum = 0;

5183   orptr = mp->b_rptr;
5184   mp->b_rptr += shift;

5186   udpha = (udpha_t *) (orptr + iph_len);
5187   if (*spi == 0) {
5188     ASSERT((uint8_t *) ipha == orptr);
5189     udpha->uha_length = htons(plen - shift - iph_len);
5190     iph_len += sizeof (udpha_t); /* For the call to ovbcopy(). */
5191     esp_ports = 0;
5192   } else {
5193     esp_ports = *((uint32_t *) udpha);
5194     ASSERT(esp_ports != 0);
5195   }
5196   ovbcopy(orptr, orptr + shift, iph_len);
5197   if (esp_ports != 0) /* Punt up for ESP processing. */ {
5198     ipha = (ipha_t *) (orptr + shift);

5200     ira->ira_flags |= IRAF_ESP_UDP_PORTS;
5201     ira->ira_esp_udp_ports = esp_ports;
5202     ip_fanout_v4(mp, ipha, ira);
5203     return (NULL);
5204   }
5205   return (mp);
5206 }

5208 /*
5209  * Deliver a udp packet to the given conn, possibly applying ipsec policy.

```

```

5210  * Handles IPv4 and IPv6.
5211  * We are responsible for disposing of mp, such as by freemsg() or putnext()
5212  * Caller is responsible for dropping references to the conn.
5213  */
5214  void
5215  ip_fanout_udp_conn(conn_t *connp, mblk_t *mp, ipha_t *ipha, ip6_t *ip6h,
5216    ip_rcv_attr_t *ira)
5217  {
5218     ill_t      *ill = ira->ira_ill;
5219     ip_stack_t  *ipst = ill->ill_ipst;
5220     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;
5221     boolean_t   secure;
5222     iaflags_t   iraflags = ira->ira_flags;

5224     secure = iraflags & IRAF_IPSEC_SECURE;

5226     if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrld :
5227       !canputnext(connp->conn_rq)) {
5228       BUMP_MIB(ill->ill_ip_mib, udpIfStatsInOverflows);
5229       freemsg(mp);
5230       return;
5231     }

5233     if (((iraflags & IRAF_IS_IPV4) ?
5234       CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
5235       CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
5236       secure) {
5237       mp = ipsec_check_inbound_policy(mp, connp, ipha,
5238         ip6h, ira);
5239       if (mp == NULL) {
5240         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5241         /* Note that mp is NULL */
5242         ip_drop_input("ipIfStatsInDiscards", mp, ill);
5243         return;
5244       }
5245     }

5247     /*
5248      * Since this code is not used for UDP unicast we don't need a NAT_T
5249      * check. Only ip_fanout_v4 has that check.
5250      */
5251     if (ira->ira_flags & IRAF_ICMP_ERROR) {
5252       (connp->conn_recvicmp)(connp, mp, NULL, ira);
5253     } else {
5254       ill_t *rill = ira->ira_rill;

5256       BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
5257       ira->ira_ill = ira->ira_rill = NULL;
5258       /* Send it upstream */
5259       (connp->conn_rcv)(connp, mp, NULL, ira);
5260       ira->ira_ill = ill;
5261       ira->ira_rill = rill;
5262     }
5263   }

5265  /*
5266   * Fanout for UDP packets that are multicast or broadcast, and ICMP errors.
5267   * (Unicast fanout is handled in ip_input_v4.)
5268   */
5269   * If SO_REUSEADDR is set all multicast and broadcast packets
5270   * will be delivered to all conns bound to the same port.
5271   */
5272   * If there is at least one matching AF_INET receiver, then we will
5273   * ignore any AF_INET6 receivers.
5274   * In the special case where an AF_INET socket binds to 0.0.0.0/<port> and an
5275   * AF_INET6 socket binds to ::/<port>, only the AF_INET socket receives the IPv4

```

```

5276 * packets.
5277 *
5278 * Zones notes:
5279 * Earlier in ip_input on a system with multiple shared-IP zones we
5280 * duplicate the multicast and broadcast packets and send them up
5281 * with each explicit zoneid that exists on that ill.
5282 * This means that here we can match the zoneid with SO_ALLZONES being special.
5283 */
5284 void
5285 ip_fanout_udp_multi_v4(mblk_t *mp, ipha_t *ipha, uint16_t lport, uint16_t fport,
5286 ip_rcv_attr_t *ira)
5287 {
5288     ipaddr_t      laddr;
5289     in6_addr_t     v6faddr;
5290     conn_t         *connp;
5291     connf_t        *connfp;
5292     ipaddr_t       faddr;
5293     ill_t          *ill = ira->ira_ill;
5294     ip_stack_t     *ipst = ill->ill_ipst;

5296     ASSERT(ira->ira_flags & (IRAF_MULTIBROADCAST|IRAF_ICMP_ERROR));

5298     laddr = ipha->ipha_dst;
5299     faddr = ipha->ipha_src;

5301     connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
5302     mutex_enter(&connfp->connf_lock);
5303     connp = connfp->connf_head;

5305     /*
5306      * If SO_REUSEADDR has been set on the first we send the
5307      * packet to all clients that have joined the group and
5308      * match the port.
5309      */
5310     while (connp != NULL) {
5311         if ((IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr)) &&
5312             conn_wantpacket(connp, ira, ipha) &&
5313             (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5314              tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp)))
5315             break;
5316         connp = connp->conn_next;
5317     }

5319     if (connp == NULL)
5320         goto notfound;

5322     CONN_INC_REF(connp);

5324     if (connp->conn_reuseaddr) {
5325         conn_t      *first_connp = connp;
5326         conn_t      *next_connp;
5327         mblk_t      *mpl;

5329         connp = connp->conn_next;
5330         for (;;) {
5331             while (connp != NULL) {
5332                 if (IPCL_UDP_MATCH(connp, lport, laddr,
5333                     fport, faddr) &&
5334                     conn_wantpacket(connp, ira, ipha) &&
5335                     (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5336                      tsol_receive_local(mp, &laddr, IPV4_VERSION,
5337                      ira, connp)))
5338                     break;
5339                 connp = connp->conn_next;
5340             }
5341             if (connp == NULL) {

```

```

5342             /* No more interested clients */
5343             connp = first_connp;
5344             break;
5345         }
5346         if (((mpl = dupmsg(mp)) == NULL) &&
5347             ((mpl = copymsg(mp)) == NULL)) {
5348             /* Memory allocation failed */
5349             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5350             ip_drop_input("ipIfStatsInDiscards", mp, ill);
5351             connp = first_connp;
5352             break;
5353         }
5354         CONN_INC_REF(connp);
5355         mutex_exit(&connfp->connf_lock);

5357         IP_STAT(ipst, ip_udp_fanmb);
5358         ip_fanout_udp_conn(connp, mpl, (ipha_t *)mpl->b_rptr,
5359             NULL, ira);
5360         mutex_enter(&connfp->connf_lock);
5361         /* Follow the next pointer before releasing the conn */
5362         next_connp = connp->conn_next;
5363         CONN_DEC_REF(connp);
5364         connp = next_connp;
5365     }
5366 }

5368 /* Last one. Send it upstream. */
5369 mutex_exit(&connfp->connf_lock);
5370 IP_STAT(ipst, ip_udp_fanmb);
5371 ip_fanout_udp_conn(connp, mp, ipha, NULL, ira);
5372 CONN_DEC_REF(connp);
5373 return;

5375 notfound:
5376 mutex_exit(&connfp->connf_lock);
5377 /*
5378  * IPv6 endpoints bound to multicast IPv4-mapped addresses
5379  * have already been matched above, since they live in the IPv4
5380  * fanout tables. This implies we only need to
5381  * check for IPv6 in6addr any endpoints here.
5382  * Thus we compare using ipv6_all_zeros instead of the destination
5383  * address, except for the multicast group membership lookup which
5384  * uses the IPv4 destination.
5385  */
5386 IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &v6faddr);
5387 connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
5388 mutex_enter(&connfp->connf_lock);
5389 connp = connfp->connf_head;
5390 /*
5391  * IPv4 multicast packet being delivered to an AF_INET6
5392  * in6addr any endpoint.
5393  * Need to check conn_wantpacket(). Note that we use conn_wantpacket()
5394  * and not conn_wantpacket_v6() since any multicast membership is
5395  * for an IPv4-mapped multicast address.
5396  */
5397 while (connp != NULL) {
5398     if (IPCL_UDP_MATCH_V6(connp, lport, ipv6_all_zeros,
5399         fport, v6faddr) &&
5400         conn_wantpacket(connp, ira, ipha) &&
5401         (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5402          tsol_receive_local(mp, &laddr, IPV4_VERSION, ira, connp)))
5403         break;
5404     connp = connp->conn_next;
5405 }

5407 if (connp == NULL) {

```

```

5408      /*
5409       * No one bound to this port. Is
5410       * there a client that wants all
5411       * unclaimed datagrams?
5412       */
5413      mutex_exit(&connfp->connf_lock);

5415      if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_UDP].connf_head !=
5416          NULL) {
5417          ASSERT(ira->ira_protocol == IPPROTO_UDP);
5418          ip_fanout_proto_v4(mp, ipha, ira);
5419      } else {
5420          /*
5421           * We used to attempt to send an icmp error here, but
5422           * since this is known to be a multicast packet
5423           * and we don't send icmp errors in response to
5424           * multicast, just drop the packet and give up sooner.
5425           */
5426          BUMP_MIB(ill->ill_ip_mib, udpIfStatsNoPorts);
5427          freemsg(mp);
5428      }
5429      return;
5430  }
5431  ASSERT(IPCL_IS_NONSTR(connp) || connp->conn_rq != NULL);

5433  /*
5434   * If SO_REUSEADDR has been set on the first we send the
5435   * packet to all clients that have joined the group and
5436   * match the port.
5437   */
5438  if (connp->conn_reuseaddr) {
5439      conn_t      *first_connp = connp;
5440      conn_t      *next_connp;
5441      mblk_t      *mpl;

5443      CONN_INC_REF(connp);
5444      connp = connp->conn_next;
5445      for (;;) {
5446          while (connp != NULL) {
5447              if (IPCL_UDP_MATCH_V6(connp, lport,
5448                  ipv6_all_zeros, fport, v6faddr) &&
5449                  conn_wantpacket(connp, ira, ipha) &&
5450                  (!(ira->ira_flags & IRAF_SYSTEM_LABELED) ||
5451                      tsol_receive_local(mp, &laddr, IPV4_VERSION,
5452                          ira, connp)))
5453                  break;
5454              connp = connp->conn_next;
5455          }
5456          if (connp == NULL) {
5457              /* No more interested clients */
5458              connp = first_connp;
5459              break;
5460          }
5461          if (((mpl = dupmsg(mp)) == NULL) &&
5462              ((mpl = copymsg(mp)) == NULL)) {
5463              /* Memory allocation failed */
5464              BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
5465              ip_drop_input("ipIfStatsInDiscards", mp, ill);
5466              connp = first_connp;
5467              break;
5468          }
5469          CONN_INC_REF(connp);
5470          mutex_exit(&connfp->connf_lock);

5472          IP_STAT(ipst, ip_udp_fanmb);
5473          ip_fanout_udp_conn(connp, mpl, (ipha_t *)mpl->b_rptr,

```

```

5474          NULL, ira);
5475          mutex_enter(&connfp->connf_lock);
5476          /* Follow the next pointer before releasing the conn */
5477          next_connp = connp->conn_next;
5478          CONN_DEC_REF(connp);
5479          connp = next_connp;
5480      }
5481  }

5483  /* Last one. Send it upstream. */
5484  mutex_exit(&connfp->connf_lock);
5485  IP_STAT(ipst, ip_udp_fanmb);
5486  ip_fanout_udp_conn(connp, mp, ipha, NULL, ira);
5487  CONN_DEC_REF(connp);
5488  }

5490  /*
5491   * Split an incoming packet's IPv4 options into the label and the other options.
5492   * If 'allocate' is set it does memory allocation for the ip_pkt_t, including
5493   * clearing out any leftover label or options.
5494   * Otherwise it just makes ipp point into the packet.
5495   *
5496   * Returns zero if ok; ENOMEM if the buffer couldn't be allocated.
5497   */
5498  int
5499  ip_find_hdr_v4(ipha_t *ipha, ip_pkt_t *ipp, boolean_t allocate)
5500  {
5501      uchar_t      *opt;
5502      uint32_t      totallen;
5503      uint32_t      optval;
5504      uint32_t      optlen;

5506      ipp->ipp_fields |= IPPF_HOPLIMIT | IPPF_TCLASS | IPPF_ADDR;
5507      ipp->ipp_hoplimit = ipha->ipha_ttl;
5508      ipp->ipp_type_of_service = ipha->ipha_type_of_service;
5509      IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &ipp->ipp_addr);

5511      /*
5512       * Get length (in 4 byte octets) of IP header options.
5513       */
5514      totallen = ipha->ipha_version_and_hdr_length -
5515          (uint8_t)((IP_VERSION < 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);

5517      if (totallen == 0) {
5518          if (!allocate)
5519              return (0);

5521          /* Clear out anything from a previous packet */
5522          if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
5523              kmem_free(ipp->ipp_ipv4_options,
5524                  ipp->ipp_ipv4_options_len);
5525              ipp->ipp_ipv4_options = NULL;
5526              ipp->ipp_ipv4_options_len = 0;
5527              ipp->ipp_fields &= ~IPPF_IPV4_OPTIONS;
5528          }
5529          if (ipp->ipp_fields & IPPF_LABEL_V4) {
5530              kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
5531              ipp->ipp_label_v4 = NULL;
5532              ipp->ipp_label_len_v4 = 0;
5533              ipp->ipp_fields &= ~IPPF_LABEL_V4;
5534          }
5535          return (0);
5536      }

5538      totallen <= 2;
5539      opt = (uchar_t *)&ipha[1];

```

```

5540     if (!is_system_labeled()) {
5542         copyall:
5543             if (!allocate) {
5544                 if (totallen != 0) {
5545                     ipp->ipp_ipv4_options = opt;
5546                     ipp->ipp_ipv4_options_len = totallen;
5547                     ipp->ipp_fields |= IPPF_IPV4_OPTIONS;
5548                 }
5549                 return (0);
5550             }
5551             /* Just copy all of options */
5552             if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
5553                 if (totallen == ipp->ipp_ipv4_options_len) {
5554                     bcopy(opt, ipp->ipp_ipv4_options, totallen);
5555                     return (0);
5556                 }
5557                 kmem_free(ipp->ipp_ipv4_options,
5558                     ipp->ipp_ipv4_options_len);
5559                 ipp->ipp_ipv4_options = NULL;
5560                 ipp->ipp_ipv4_options_len = 0;
5561                 ipp->ipp_fields &= ~IPPF_IPV4_OPTIONS;
5562             }
5563             if (totallen == 0)
5564                 return (0);

5566             ipp->ipp_ipv4_options = kmem_alloc(totallen, KM_NOSLEEP);
5567             if (ipp->ipp_ipv4_options == NULL)
5568                 return (ENOMEM);
5569             ipp->ipp_ipv4_options_len = totallen;
5570             ipp->ipp_fields |= IPPF_IPV4_OPTIONS;
5571             bcopy(opt, ipp->ipp_ipv4_options, totallen);
5572             return (0);
5573         }

5575     if (allocate && (ipp->ipp_fields & IPPF_LABEL_V4)) {
5576         kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
5577         ipp->ipp_label_v4 = NULL;
5578         ipp->ipp_label_len_v4 = 0;
5579         ipp->ipp_fields &= ~IPPF_LABEL_V4;
5580     }

5582     /*
5583     * Search for CIPSO option.
5584     * We assume CIPSO is first in options if it is present.
5585     * If it isn't, then ipp_opt_ipv4_options will not include the options
5586     * prior to the CIPSO option.
5587     */
5588     while (totallen != 0) {
5589         switch (optval = opt[IPOPT_OPTVAL]) {
5590             case IPOPT_EOL:
5591                 return (0);
5592             case IPOPT_NOP:
5593                 optlen = 1;
5594                 break;
5595             default:
5596                 if (totallen <= IPOPT_OLEN)
5597                     return (EINVAL);
5598                 optlen = opt[IPOPT_OLEN];
5599                 if (optlen < 2)
5600                     return (EINVAL);
5601             }
5602             if (optlen > totallen)
5603                 return (EINVAL);

5605             switch (optval) {

```

```

5606             case IPOPT_COMSEC:
5607                 if (!allocate) {
5608                     ipp->ipp_label_v4 = opt;
5609                     ipp->ipp_label_len_v4 = optlen;
5610                     ipp->ipp_fields |= IPPF_LABEL_V4;
5611                 } else {
5612                     ipp->ipp_label_v4 = kmem_alloc(optlen,
5613                         KM_NOSLEEP);
5614                     if (ipp->ipp_label_v4 == NULL)
5615                         return (ENOMEM);
5616                     ipp->ipp_label_len_v4 = optlen;
5617                     ipp->ipp_fields |= IPPF_LABEL_V4;
5618                     bcopy(opt, ipp->ipp_label_v4, optlen);
5619                 }
5620                 totallen -= optlen;
5621                 opt += optlen;

5623                 /* Skip padding bytes until we get to a multiple of 4 */
5624                 while ((totallen & 3) != 0 && opt[0] == IPOPT_NOP) {
5625                     totallen--;
5626                     opt++;
5627                 }
5628                 /* Remaining as ipp_ipv4_options */
5629                 goto copyall;
5630             }
5631             totallen -= optlen;
5632             opt += optlen;
5633         }
5634         /* No CIPSO found; return everything as ipp_ipv4_options */
5635         totallen = ipha->ipha_version_and_hdr_length -
5636             (uint8_t)((IP_VERSION << 4) + IP_SIMPLE_HDR_LENGTH_IN_WORDS);
5637         totallen <= 2;
5638         opt = (uchar_t *)&ipha[1];
5639         goto copyall;
5640     }

5642     /*
5643     * Efficient versions of lookup for an IRE when we only
5644     * match the address.
5645     * For RTF_REJECT or BLACKHOLE we return IRE_NOROUTE.
5646     * Does not handle multicast addresses.
5647     */
5648     uint_t
5649     ip_type_v4(ipaddr_t addr, ip_stack_t *ipst)
5650     {
5651         ire_t *ire;
5652         uint_t result;

5654         ire = ire_fhtable_lookup_simple_v4(addr, 0, ipst, NULL);
5655         ASSERT(ire != NULL);
5656         if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))
5657             result = IRE_NOROUTE;
5658         else
5659             result = ire->ire_type;
5660         ire_refrele(ire);
5661         return (result);
5662     }

5664     /*
5665     * Efficient versions of lookup for an IRE when we only
5666     * match the address.
5667     * For RTF_REJECT or BLACKHOLE we return IRE_NOROUTE.
5668     * Does not handle multicast addresses.
5669     */
5670     uint_t
5671     ip_type_v6(const in6_addr_t *addr, ip_stack_t *ipst)

```

```

5672 {
5673     ire_t *ire;
5674     uint_t result;

5676     ire = ire_fhtable_lookup_simple_v6(addr, 0, ipst, NULL);
5677     ASSERT(ire != NULL);
5678     if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE))
5679         result = IRE_NOROUTE;
5680     else
5681         result = ire->ire_type;
5682     ire_refrele(ire);
5683     return (result);
5684 }

5686 /*
5687  * Nobody should be sending
5688  * packets up this stream
5689  */
5690 static void
5691 ip_lrput(queue_t *q, mblk_t *mp)
5692 {
5693     switch (mp->b_datap->db_type) {
5694     case M_FLUSH:
5695         /* Turn around */
5696         if (*mp->b_rptr & FLUSHW) {
5697             *mp->b_rptr &= ~FLUSHR;
5698             greply(q, mp);
5699             return;
5700         }
5701         break;
5702     }
5703     freemsg(mp);
5704 }

5706 /* Nobody should be sending packets down this stream */
5707 /* ARGSUSED */
5708 void
5709 ip_lwput(queue_t *q, mblk_t *mp)
5710 {
5711     freemsg(mp);
5712 }

5714 /*
5715  * Move the first hop in any source route to ipha_dst and remove that part of
5716  * the source route. Called by other protocols. Errors in option formatting
5717  * are ignored - will be handled by ip_output_options. Return the final
5718  * destination (either ipha_dst or the last entry in a source route.)
5719  */
5720 ipaddr_t
5721 ip_message_options(ipha_t *ipha, netstack_t *ns)
5722 {
5723     ipoptp_t      opts;
5724     uchar_t       *opt;
5725     uint8_t       optval;
5726     uint8_t       optlen;
5727     ipaddr_t      dst;
5728     int           i;
5729     ip_stack_t    *ipst = ns->netstack_ip;

5731     ip2dbg(("ip_message_options\n"));
5732     dst = ipha->ipha_dst;
5733     for (optval = ipoptp_first(&opts, ipha);
5734          optval != IPOPT_EOL;
5735          optval = ipoptp_next(&opts)) {
5736         opt = opts.ipoptp_cur;
5737         switch (optval) {

```

```

5738         uint8_t off;
5739     case IPOPT_SSRR:
5740     case IPOPT_LSRR:
5741         if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
5742             ipldbg(("ip_message_options: bad src route\n"));
5743             break;
5744         }
5745         optlen = opts.ipoptp_len;
5746         off = opt[IPOPT_OFFSET];
5747         off--;
5748     redo_srr:
5749         if (optlen < IP_ADDR_LEN ||
5750             off > optlen - IP_ADDR_LEN) {
5751             /* End of source route */
5752             ipldbg(("ip_message_options: end of SR\n"));
5753             break;
5754         }
5755         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
5756         ipldbg(("ip_message_options: next hop 0x%x\n",
5757             ntohl(dst)));
5758         /*
5759          * Check if our address is present more than
5760          * once as consecutive hops in source route.
5761          * XXX verify per-interface ip_forwarding
5762          * for source route?
5763          */
5764         if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
5765             off += IP_ADDR_LEN;
5766             goto redo_srr;
5767         }
5768         if (dst == htonl(INADDR_LOOPBACK)) {
5769             ipldbg(("ip_message_options: loopback addr in "
5770                 "source route!\n"));
5771             break;
5772         }
5773         /*
5774          * Update ipha_dst to be the first hop and remove the
5775          * first hop from the source route (by overwriting
5776          * part of the option with NOP options).
5777          */
5778         ipha->ipha_dst = dst;
5779         /* Put the last entry in dst */
5780         off = ((optlen - IP_ADDR_LEN - 3) & ~(IP_ADDR_LEN-1)) +
5781             3;
5782         bcopy(&opt[off], &dst, IP_ADDR_LEN);

5784         ipldbg(("ip_message_options: last hop 0x%x\n",
5785             ntohl(dst)));
5786         /* Move down and overwrite */
5787         opt[IP_ADDR_LEN] = opt[0];
5788         opt[IP_ADDR_LEN+1] = opt[IPOPT_OLEN] - IP_ADDR_LEN;
5789         opt[IP_ADDR_LEN+2] = opt[IPOPT_OFFSET];
5790         for (i = 0; i < IP_ADDR_LEN; i++)
5791             opt[i] = IPOPT_NOP;
5792         break;
5793     }
5794 }
5795 return (dst);
5796 }

5798 /*
5799  * Return the network mask
5800  * associated with the specified address.
5801  */
5802 ipaddr_t
5803 ip_net_mask(ipaddr_t addr)

```

```

5804 {
5805     uchar_t *up = (uchar_t *)&addr;
5806     ipaddr_t mask = 0;
5807     uchar_t *maskp = (uchar_t *)&mask;

5809 #if defined(__i386) || defined(__amd64)
5810 #define TOTALLY_BRAIN_DAMAGED_C_COMPILER
5811 #endif
5812 #ifndef TOTALLY_BRAIN_DAMAGED_C_COMPILER
5813     maskp[0] = maskp[1] = maskp[2] = maskp[3] = 0;
5814 #endif
5815     if (CLASSD(addr)) {
5816         maskp[0] = 0xF0;
5817         return (mask);
5818     }

5820     /* We assume Class E default netmask to be 32 */
5821     if (CLASSE(addr))
5822         return (0xffffffffU);

5824     if (addr == 0)
5825         return (0);
5826     maskp[0] = 0xFF;
5827     if ((up[0] & 0x80) == 0)
5828         return (mask);

5830     maskp[1] = 0xFF;
5831     if ((up[0] & 0xC0) == 0x80)
5832         return (mask);

5834     maskp[2] = 0xFF;
5835     if ((up[0] & 0xE0) == 0xC0)
5836         return (mask);

5838     /* Otherwise return no mask */
5839     return ((ipaddr_t)0);
5840 }

5842 /* Name/Value Table Lookup Routine */
5843 char *
5844 ip_nv_lookup(nv_t *nv, int value)
5845 {
5846     if (!nv)
5847         return (NULL);
5848     for (; nv->nv_name; nv++) {
5849         if (nv->nv_value == value)
5850             return (nv->nv_name);
5851     }
5852     return ("unknown");
5853 }

5855 static int
5856 ip_wait_for_info_ack(ill_t *ill)
5857 {
5858     int err;

5860     mutex_enter(&ill->ill_lock);
5861     while (ill->ill_state_flags & ILL_LL_SUBNET_PENDING) {
5862         /*
5863          * Return value of 0 indicates a pending signal.
5864          */
5865         err = cv_wait_sig(&ill->ill_cv, &ill->ill_lock);
5866         if (err == 0) {
5867             mutex_exit(&ill->ill_lock);
5868             return (EINTR);
5869         }

```

```

5870     }
5871     mutex_exit(&ill->ill_lock);
5872     /*
5873      * ip_rput_other could have set an error in ill_error on
5874      * receipt of M_ERROR.
5875      */
5876     return (ill->ill_error);
5877 }

5879 /*
5880  * This is a module open, i.e. this is a control stream for access
5881  * to a DLPI device. We allocate an ill_t as the instance data in
5882  * this case.
5883  */
5884 static int
5885 ip_modopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5886 {
5887     ill_t *ill;
5888     int err;
5889     zoneid_t zoneid;
5890     netstack_t *ns;
5891     ip_stack_t *ipst;

5893     /*
5894      * Prevent unprivileged processes from pushing IP so that
5895      * they can't send raw IP.
5896      */
5897     if (secpolicy_net_rawaccess(credp) != 0)
5898         return (EPERM);

5900     ns = netstack_find_by_cred(credp);
5901     ASSERT(ns != NULL);
5902     ipst = ns->netstack_ip;
5903     ASSERT(ipst != NULL);

5905     /*
5906      * For exclusive stacks we set the zoneid to zero
5907      * to make IP operate as if in the global zone.
5908      */
5909     if (ipst->ips_netstack->netstack_stackid != GLOBAL_NETSTACKID)
5910         zoneid = GLOBAL_ZONEID;
5911     else
5912         zoneid = crgetzoneid(credp);

5914     ill = (ill_t *)mi_open_alloc_sleep(sizeof (ill_t));
5915     q->q_ptr = WR(q)->q_ptr = ill;
5916     ill->ill_ipst = ipst;
5917     ill->ill_zoneid = zoneid;

5919     /*
5920      * ill_init initializes the ill fields and then sends down
5921      * a DL_INFO_REQ after calling qprocson.
5922      */
5923     err = ill_init(q, ill);

5925     if (err != 0) {
5926         mi_free(ill);
5927         netstack_rele(ipst->ips_netstack);
5928         q->q_ptr = NULL;
5929         WR(q)->q_ptr = NULL;
5930         return (err);
5931     }

5933     /*
5934      * Wait for the DL_INFO_ACK if a DL_INFO_REQ was sent.
5935      */

```

```

5936     * ill_init initializes the ipsq marking this thread as
5937     * writer
5938     */
5939     ipsq_exit(ill->ill_phyint->phyint_ipsq);
5940     err = ip_wait_for_info_ack(ill);
5941     if (err == 0)
5942         ill->ill_credp = credp;
5943     else
5944         goto fail;

5946     crhold(credp);

5948     mutex_enter(&ipst->ips_ip_mi_lock);
5949     err = mi_open_link(&ipst->ips_ip_g_head, (IDP)q->q_ptr, devp, flag,
5950         sflag, credp);
5951     mutex_exit(&ipst->ips_ip_mi_lock);
5952 fail:
5953     if (err) {
5954         (void) ip_close(q, 0);
5955         return (err);
5956     }
5957     return (0);
5958 }

5960 /* For /dev/ip aka AF_INET open */
5961 int
5962 ip_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5963 {
5964     return (ip_open(q, devp, flag, sflag, credp, B_FALSE));
5965 }

5967 /* For /dev/ip6 aka AF_INET6 open */
5968 int
5969 ip_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
5970 {
5971     return (ip_open(q, devp, flag, sflag, credp, B_TRUE));
5972 }

5974 /* IP open routine. */
5975 int
5976 ip_open(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp,
5977     boolean_t isv6)
5978 {
5979     conn_t      *connp;
5980     major_t      maj;
5981     zoneid_t     zoneid;
5982     netstack_t   *ns;
5983     ip_stack_t   *ipst;

5985     /* Allow reopen. */
5986     if (q->q_ptr != NULL)
5987         return (0);

5989     if (sflag & MODOPEN) {
5990         /* This is a module open */
5991         return (ip_modopen(q, devp, flag, sflag, credp));
5992     }

5994     if ((flag & ~(FKLYR)) == IP_HELPER_STR) {
5995         /*
5996          * Non streams based socket looking for a stream
5997          * to access IP
5998          */
5999         return (ip_helper_stream_setup(q, devp, flag, sflag,
6000             credp, isv6));
6001     }

```

```

6003     ns = netstack_find_by_cred(credp);
6004     ASSERT(ns != NULL);
6005     ipst = ns->netstack_ip;
6006     ASSERT(ipst != NULL);

6008     /*
6009     * For exclusive stacks we set the zoneid to zero
6010     * to make IP operate as if in the global zone.
6011     */
6012     if (ipst->ips_netstack->netstack_stackid != GLOBAL_NETSTACKID)
6013         zoneid = GLOBAL_ZONEID;
6014     else
6015         zoneid = crgetzoneid(credp);

6017     /*
6018     * We are opening as a device. This is an IP client stream, and we
6019     * allocate a conn_t as the instance data.
6020     */
6021     connp = ipcl_conn_create(IPCL_IPCONN, KM_SLEEP, ipst->ips_netstack);

6023     /*
6024     * ipcl_conn_create did a netstack_hold. Undo the hold that was
6025     * done by netstack_find_by_cred()
6026     */
6027     netstack_rele(ipst->ips_netstack);

6029     connp->conn_ixa->ixa_flags |= IXAF_MULTICAST_LOOP | IXAF_SET_ULP_CKSUM;
6030     /* conn_allzones can not be set this early, hence no IPCL_ZONEID */
6031     connp->conn_ixa->ixa_zoneid = zoneid;
6032     connp->conn_zoneid = zoneid;

6034     connp->conn_rq = q;
6035     q->q_ptr = WR(q)->q_ptr = connp;

6037     /* Minor tells us which /dev entry was opened */
6038     if (isv6) {
6039         connp->conn_family = AF_INET6;
6040         connp->conn_ipversion = IPV6_VERSION;
6041         connp->conn_ixa->ixa_flags &= ~IXAF_IS_IPV4;
6042         connp->conn_ixa->ixa_src_preferences = IPV6_PREFER_SRC_DEFAULT;
6043     } else {
6044         connp->conn_family = AF_INET;
6045         connp->conn_ipversion = IPV4_VERSION;
6046         connp->conn_ixa->ixa_flags |= IXAF_IS_IPV4;
6047     }

6049     if ((ip_minor_arena_la != NULL) && (flag & SO_SOCKSTR) &&
6050         ((connp->conn_dev = inet_minor_alloc(ip_minor_arena_la)) != 0)) {
6051         connp->conn_minor_arena = ip_minor_arena_la;
6052     } else {
6053         /*
6054         * Either minor numbers in the large arena were exhausted
6055         * or a non socket application is doing the open.
6056         * Try to allocate from the small arena.
6057         */
6058         if ((connp->conn_dev =
6059             inet_minor_alloc(ip_minor_arena_sa)) == 0) {
6060             /* CONN_DEC_REF takes care of netstack_rele() */
6061             q->q_ptr = WR(q)->q_ptr = NULL;
6062             CONN_DEC_REF(connp);
6063             return (EBUSY);
6064         }
6065         connp->conn_minor_arena = ip_minor_arena_sa;
6066     }

```

```

6068     maj = getemajor(*devp);
6069     *devp = makedevice(maj, (minor_t)connp->conn_dev);

6071     /*
6072      * connp->conn_cred is crfree()ed in ipcl_conn_destroy()
6073      */
6074     connp->conn_cred = credp;
6075     connp->conn_cpid = curproc->p_pid;
6076     /* Cache things in ixa without an extra refhold */
6077     ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
6078     connp->conn_ixa->ixa_cred = connp->conn_cred;
6079     connp->conn_ixa->ixa_cpid = connp->conn_cpid;
6080     if (is_system_labeled())
6081         connp->conn_ixa->ixa_tsl = crgetlabel(connp->conn_cred);

6083     /*
6084      * Handle IP_IOC_RTS_REQUEST and other ioctls which use conn_rcv
6085      */
6086     connp->conn_rcv = ip_conn_input;
6087     connp->conn_rcvicmp = ip_conn_input_icmp;

6089     crhold(connp->conn_cred);

6091     /*
6092      * If the caller has the process-wide flag set, then default to MAC
6093      * exempt mode. This allows read-down to unlabeled hosts.
6094      */
6095     if (getpflags(NET_MAC_AWARE, credp) != 0)
6096         connp->conn_mac_mode = CONN_MAC_AWARE;

6098     connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);

6100     connp->conn_rq = q;
6101     connp->conn_wq = WR(q);

6103     /* Non-zero default values */
6104     connp->conn_ixa->ixa_flags |= IXAF_MULTICAST_LOOP;

6106     /*
6107      * Make the conn globally visible to walkers
6108      */
6109     ASSERT(connp->conn_ref == 1);
6110     mutex_enter(&connp->conn_lock);
6111     connp->conn_state_flags &= ~CONN_INCIPIENT;
6112     mutex_exit(&connp->conn_lock);

6114     qprocson(q);

6116     return (0);
6117 }

6119 /*
6120  * Set IPsec policy from an ipsec_req_t. If the req is not "zero" and valid,
6121  * all of them are copied to the conn_t. If the req is "zero", the policy is
6122  * zeroed out. A "zero" policy has zero ipshr_{ah,req,self_encap}_req
6123  * fields.
6124  * We keep only the latest setting of the policy and thus policy setting
6125  * is not incremental/cumulative.
6126  * Requests to set policies with multiple alternative actions will
6127  * go through a different API.
6128  */
6129 int
6130 ipsec_set_req(cred_t *cr, conn_t *connp, ipsec_req_t *req)
6131 {
6132     uint_t ah_req = 0;

```

```

6134     uint_t esp_req = 0;
6135     uint_t se_req = 0;
6136     ipsec_act_t *actp = NULL;
6137     uint_t nact;
6138     ipsec_policy_head_t *ph;
6139     boolean_t is_pol_reset, is_pol_inserted = B_FALSE;
6140     int error = 0;
6141     netstack_t *ns = connp->conn_netstack;
6142     ip_stack_t *ipst = ns->netstack_ip;
6143     ipsec_stack_t *ipss = ns->netstack_ipsec;

6145 #define REQ_MASK (IPSEC_PREF_REQUIRED|IPSEC_PREF_NEVER)

6147     /*
6148      * The IP_SEC_OPT option does not allow variable length parameters,
6149      * hence a request cannot be NULL.
6150      */
6151     if (req == NULL)
6152         return (EINVAL);

6154     ah_req = req->ipshr_ah_req;
6155     esp_req = req->ipshr_esp_req;
6156     se_req = req->ipshr_self_encap_req;

6158     /* Don't allow setting self-encap without one or more of AH/ESP. */
6159     if (se_req != 0 && esp_req == 0 && ah_req == 0)
6160         return (EINVAL);

6162     /*
6163      * Are we dealing with a request to reset the policy (i.e.
6164      * zero requests).
6165      */
6166     is_pol_reset = ((ah_req & REQ_MASK) == 0 &&
6167         (esp_req & REQ_MASK) == 0 &&
6168         (se_req & REQ_MASK) == 0);

6170     if (!is_pol_reset) {
6171         /*
6172          * If we couldn't load IPsec, fail with "protocol
6173          * not supported".
6174          * IPsec may not have been loaded for a request with zero
6175          * policies, so we don't fail in this case.
6176          */
6177         mutex_enter(&ipss->ipsec_loader_lock);
6178         if (ipss->ipsec_loader_state != IPSEC_LOADER_SUCCEEDED) {
6179             mutex_exit(&ipss->ipsec_loader_lock);
6180             return (EPROTONOSUPPORT);
6181         }
6182         mutex_exit(&ipss->ipsec_loader_lock);

6184         /*
6185          * Test for valid requests. Invalid algorithms
6186          * need to be tested by IPsec code because new
6187          * algorithms can be added dynamically.
6188          */
6189         if ((ah_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0 ||
6190             (esp_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0 ||
6191             (se_req & ~(REQ_MASK|IPSEC_PREF_UNIQUE)) != 0) {
6192             return (EINVAL);
6193         }

6195         /*
6196          * Only privileged users can issue these
6197          * requests.
6198          */
6199         if (((ah_req & IPSEC_PREF_NEVER) ||

```



```

6200         (esp_req & IPSEC_PREF_NEVER) ||
6201         (se_req & IPSEC_PREF_NEVER)) &&
6202         secpolicy_ip_config(cr, B_FALSE) != 0) {
6203             return (EPERM);
6204         }
6205
6206     /*
6207     * The IPSEC_PREF_REQUIRED and IPSEC_PREF_NEVER
6208     * are mutually exclusive.
6209     */
6210     if (((ah_req & REQ_MASK) == REQ_MASK) ||
6211         ((esp_req & REQ_MASK) == REQ_MASK) ||
6212         ((se_req & REQ_MASK) == REQ_MASK)) {
6213         /* Both of them are set */
6214         return (EINVAL);
6215     }
6216 }
6217
6218 ASSERT(MUTEX_HELD(&connp->conn_lock));
6219
6220 /*
6221 * If we have already cached policies in conn_connect(), don't
6222 * let them change now. We cache policies for connections
6223 * whose src,dst [addr, port] is known.
6224 */
6225 if (connp->conn_policy_cached) {
6226     return (EINVAL);
6227 }
6228
6229 /*
6230 * We have a zero policies, reset the connection policy if already
6231 * set. This will cause the connection to inherit the
6232 * global policy, if any.
6233 */
6234 if (is_pol_reset) {
6235     if (connp->conn_policy != NULL) {
6236         IPPH_REFRELE(connp->conn_policy, ipst->ips_netstack);
6237         connp->conn_policy = NULL;
6238     }
6239     connp->conn_in_enforce_policy = B_FALSE;
6240     connp->conn_out_enforce_policy = B_FALSE;
6241     return (0);
6242 }
6243
6244 ph = connp->conn_policy = ipsec_polhead_split(connp->conn_policy,
6245     ipst->ips_netstack);
6246 if (ph == NULL)
6247     goto enomem;
6248
6249 ipsec_actvec_from_req(req, &actp, &nact, ipst->ips_netstack);
6250 if (actp == NULL)
6251     goto enomem;
6252
6253 /*
6254 * Always insert IPv4 policy entries, since they can also apply to
6255 * ipv6 sockets being used in ipv4-compat mode.
6256 */
6257 if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V4,
6258     IPSEC_TYPE_INBOUND, ns))
6259     goto enomem;
6260 is_pol_inserted = B_TRUE;
6261 if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V4,
6262     IPSEC_TYPE_OUTBOUND, ns))
6263     goto enomem;
6264
6265 /*

```

```

6266     * We're looking at a v6 socket, also insert the v6-specific
6267     * entries.
6268     */
6269     if (connp->conn_family == AF_INET6) {
6270         if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V6,
6271             IPSEC_TYPE_INBOUND, ns))
6272             goto enomem;
6273         if (!ipsec_polhead_insert(ph, actp, nact, IPSEC_AF_V6,
6274             IPSEC_TYPE_OUTBOUND, ns))
6275             goto enomem;
6276     }
6277
6278     ipsec_actvec_free(actp, nact);
6279
6280     /*
6281     * If the requests need security, set enforce_policy.
6282     * If the requests are IPSEC_PREF_NEVER, one should
6283     * still set conn_out_enforce_policy so that ip_set_destination
6284     * marks the ip_xmit_attr_t appropriately. This is needed so that
6285     * for connections that we don't cache policy in at connect time,
6286     * if global policy matches in ip_output_attach_policy, we
6287     * don't wrongly inherit global policy. Similarly, we need
6288     * to set conn_in_enforce_policy also so that we don't verify
6289     * policy wrongly.
6290     */
6291     if ((ah_req & REQ_MASK) != 0 ||
6292         (esp_req & REQ_MASK) != 0 ||
6293         (se_req & REQ_MASK) != 0) {
6294         connp->conn_in_enforce_policy = B_TRUE;
6295         connp->conn_out_enforce_policy = B_TRUE;
6296     }
6297
6298     return (error);
6299 #undef REQ_MASK
6300
6301     /*
6302     * Common memory-allocation-failure exit path.
6303     */
6304     enomem:
6305     if (actp != NULL)
6306         ipsec_actvec_free(actp, nact);
6307     if (is_pol_inserted)
6308         ipsec_polhead_flush(ph, ns);
6309     return (ENOMEM);
6310 }
6311
6312 /*
6313 * Set socket options for joining and leaving multicast groups.
6314 * Common to IPv4 and IPv6; inet6 indicates the type of socket.
6315 * The caller has already check that the option name is consistent with
6316 * the address family of the socket.
6317 */
6318 int
6319 ip_opt_set_multicast_group(conn_t *connp, t_scalar_t name,
6320     uchar_t *invalp, boolean_t inet6, boolean_t checkonly)
6321 {
6322     int             *il = (int *)invalp;
6323     int             error = 0;
6324     ip_stack_t      *ipst = connp->conn_netstack->netstack_ip;
6325     struct ip_mreq  *v4_mreq;
6326     struct ipv6_mreq *v6_mreq;
6327     struct group_req *gregp;
6328     ire_t *ire;
6329     boolean_t done = B_FALSE;
6330     ipaddr_t ifaddr;
6331     in6_addr_t v6group;

```

```

6332     uint_t ifindex;
6333     boolean_t mcast_opt = B_TRUE;
6334     mcast_record_t fmode;
6335     int (*optfn)(conn_t *, boolean_t, const in6_addr_t *,
6336         ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *);

6338     switch (name) {
6339     case IP_ADD_MEMBERSHIP:
6340     case IPV6_JOIN_GROUP:
6341         mcast_opt = B_FALSE;
6342         /* FALLTHRU */
6343     case MCAST_JOIN_GROUP:
6344         fmode = MODE_IS_EXCLUDE;
6345         optfn = ip_opt_add_group;
6346         break;

6348     case IP_DROP_MEMBERSHIP:
6349     case IPV6_LEAVE_GROUP:
6350         mcast_opt = B_FALSE;
6351         /* FALLTHRU */
6352     case MCAST_LEAVE_GROUP:
6353         fmode = MODE_IS_INCLUDE;
6354         optfn = ip_opt_delete_group;
6355         break;
6356     default:
6357         ASSERT(0);
6358     }

6360     if (mcast_opt) {
6361         struct sockaddr_in *sin;
6362         struct sockaddr_in6 *sin6;

6364         greqp = (struct group_req *)il;
6365         if (greqp->gr_group.ss_family == AF_INET) {
6366             sin = (struct sockaddr_in *)&(greqp->gr_group);
6367             IN6_INADDR_TO_V4MAPPED(&sin->sin_addr, &v6group);
6368         } else {
6369             if (!inet6)
6370                 return (EINVAL); /* Not on INET socket */

6372             sin6 = (struct sockaddr_in6 *)&(greqp->gr_group);
6373             v6group = sin6->sin6_addr;
6374         }
6375         ifaddr = INADDR_ANY;
6376         ifindex = greqp->gr_interface;
6377     } else if (inet6) {
6378         v6_mreqp = (struct ipv6_mreq *)il;
6379         v6group = v6_mreqp->ipv6mr_multiaddr;
6380         ifaddr = INADDR_ANY;
6381         ifindex = v6_mreqp->ipv6mr_interface;
6382     } else {
6383         v4_mreqp = (struct ip_mreq *)il;
6384         IN6_INADDR_TO_V4MAPPED(&v4_mreqp->imr_multiaddr, &v6group);
6385         ifaddr = (ipaddr_t)v4_mreqp->imr_interface.s_addr;
6386         ifindex = 0;
6387     }

6389     /*
6390     * In the multirouting case, we need to replicate
6391     * the request on all interfaces that will take part
6392     * in replication. We do so because multirouting is
6393     * reflective, thus we will probably receive multi-
6394     * casts on those interfaces.
6395     * The ip_multirt_apply_membership() succeeds if
6396     * the operation succeeds on at least one interface.
6397     */

```

```

6398     if (IN6_IS_ADDR_V4MAPPED(&v6group)) {
6399         ipaddr_t group;

6401         IN6_V4MAPPED_TO_IPADDR(&v6group, group);

6403         ire = ire_fhtable_lookup_v4(group, IP_HOST_MASK, 0,
6404             IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6405             MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6406     } else {
6407         ire = ire_fhtable_lookup_v6(&v6group, &ipv6_all_ones, 0,
6408             IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6409             MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6410     }
6411     if (ire != NULL) {
6412         if (ire->ire_flags & RTF_MULTIRT) {
6413             error = ip_multirt_apply_membership(optfn, ire, connp,
6414                 checkonly, &v6group, fmode, &ipv6_all_zeros);
6415             done = B_TRUE;
6416         }
6417         ire_refrele(ire);
6418     }

6420     if (!done) {
6421         error = optfn(connp, checkonly, &v6group, ifaddr, ifindex,
6422             fmode, &ipv6_all_zeros);
6423     }
6424     return (error);
6425 }

6427 /*
6428  * Set socket options for joining and leaving multicast groups
6429  * for specific sources.
6430  * Common to IPv4 and IPv6; inet6 indicates the type of socket.
6431  * The caller has already check that the option name is consistent with
6432  * the address family of the socket.
6433  */
6434 int
6435 ip_opt_set_multicast_sources(conn_t *connp, t_scalar_t name,
6436     uchar_t *invalp, boolean_t inet6, boolean_t checkonly)
6437 {
6438     int *il = (int *)invalp;
6439     int error = 0;
6440     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;
6441     struct ip_mreq_source *imreqp;
6442     struct group_source_req *gsreqp;
6443     in6_addr_t v6group, v6src;
6444     uint32_t ifindex;
6445     ipaddr_t ifaddr;
6446     boolean_t mcast_opt = B_TRUE;
6447     mcast_record_t fmode;
6448     ire_t *ire;
6449     boolean_t done = B_FALSE;
6450     int (*optfn)(conn_t *, boolean_t, const in6_addr_t *,
6451         ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *);

6453     switch (name) {
6454     case IP_BLOCK_SOURCE:
6455         mcast_opt = B_FALSE;
6456         /* FALLTHRU */
6457     case MCAST_BLOCK_SOURCE:
6458         fmode = MODE_IS_EXCLUDE;
6459         optfn = ip_opt_add_group;
6460         break;

6462     case IP_UNBLOCK_SOURCE:
6463         mcast_opt = B_FALSE;

```

```

6464         /* FALLTHRU */
6465     case MCAST_UNBLOCK_SOURCE:
6466         fmode = MODE_IS_EXCLUDE;
6467         optfn = ip_opt_delete_group;
6468         break;
6470
6471     case IP_ADD_SOURCE_MEMBERSHIP:
6472         mcast_opt = B_FALSE;
6473         /* FALLTHRU */
6474     case MCAST_JOIN_SOURCE_GROUP:
6475         fmode = MODE_IS_INCLUDE;
6476         optfn = ip_opt_add_group;
6477         break;
6478
6479     case IP_DROP_SOURCE_MEMBERSHIP:
6480         mcast_opt = B_FALSE;
6481         /* FALLTHRU */
6482     case MCAST_LEAVE_SOURCE_GROUP:
6483         fmode = MODE_IS_INCLUDE;
6484         optfn = ip_opt_delete_group;
6485         break;
6486     default:
6487         ASSERT(0);
6488     }
6489
6490     if (mcast_opt) {
6491         gsreqp = (struct group_source_req *)il;
6492         ifindex = gsreqp->gsr_interface;
6493         if (gsreqp->gsr_group.ss_family == AF_INET) {
6494             struct sockaddr_in *s;
6495             s = (struct sockaddr_in *)&gsreqp->gsr_group;
6496             IN6_INADDR_TO_V4MAPPED(&s->sin_addr, &v6group);
6497             s = (struct sockaddr_in *)&gsreqp->gsr_source;
6498             IN6_INADDR_TO_V4MAPPED(&s->sin_addr, &v6src);
6499         } else {
6500             struct sockaddr_in6 *s6;
6501
6502             if (!inet6)
6503                 return (EINVAL); /* Not on INET socket */
6504
6505             s6 = (struct sockaddr_in6 *)&gsreqp->gsr_group;
6506             v6group = s6->sin6_addr;
6507             s6 = (struct sockaddr_in6 *)&gsreqp->gsr_source;
6508             v6src = s6->sin6_addr;
6509         }
6510         ifaddr = INADDR_ANY;
6511     } else {
6512         imreqp = (struct ip_mreq_source *)il;
6513         IN6_INADDR_TO_V4MAPPED(&imreqp->imr_multiaddr, &v6group);
6514         IN6_INADDR_TO_V4MAPPED(&imreqp->imr_sourceaddr, &v6src);
6515         ifaddr = (ipaddr_t)imreqp->imr_interface.s_addr;
6516         ifindex = 0;
6517     }
6518
6519     /*
6520      * Handle src being mapped INADDR_ANY by changing it to unspecified.
6521      */
6522     if (IN6_IS_ADDR_V4MAPPED_ANY(&v6src))
6523         v6src = ipv6_all_zeros;
6524
6525     /*
6526      * In the multirouting case, we need to replicate
6527      * the request as noted in the mcast cases above.
6528      */
6529     if (IN6_IS_ADDR_V4MAPPED(&v6group)) {
6530         ipaddr_t group;

```

```

6531         IN6_V4MAPPED_TO_IPADDR(&v6group, group);
6532
6533         ire = ire_fhtable_lookup_v4(group, IP_HOST_MASK, 0,
6534             IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6535             MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6536     } else {
6537         ire = ire_fhtable_lookup_v6(&v6group, &ipv6_all_ones, 0,
6538             IRE_HOST | IRE_INTERFACE, NULL, ALL_ZONES, NULL,
6539             MATCH_IRE_MASK | MATCH_IRE_TYPE, 0, ipst, NULL);
6540     }
6541     if (ire != NULL) {
6542         if (ire->ire_flags & RTF_MULTIRT) {
6543             error = ip_multirt_apply_membership(optfn, ire, connp,
6544                 checkonly, &v6group, fmode, &v6src);
6545             done = B_TRUE;
6546         }
6547         ire_refrele(ire);
6548     }
6549     if (!done) {
6550         error = optfn(connp, checkonly, &v6group, ifaddr, ifindex,
6551             fmode, &v6src);
6552     }
6553     return (error);
6554 }
6555
6556 /*
6557  * Given a destination address and a pointer to where to put the information
6558  * this routine fills in the mtuinfo.
6559  * The socket must be connected.
6560  * For sctp conn_faddr is the primary address.
6561  */
6562 int
6563 ip_fill_mtuinfo(conn_t *connp, ip_xmit_attr_t *ixa, struct ip6_mtuinfo *mtuinfo)
6564 {
6565     uint32_t pmtu = IP_MAXPACKET;
6566     uint_t scopeid;
6567
6568     if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6))
6569         return (-1);
6570
6571     /* In case we never sent or called ip_set_destination_v4/v6 */
6572     if (ixa->ixa_ire != NULL)
6573         pmtu = ip_get_pmtu(ixa);
6574
6575     if (ixa->ixa_flags & IXAF_SCOPEID_SET)
6576         scopeid = ixaf_scopeid;
6577     else
6578         scopeid = 0;
6579
6580     bzero(mtuinfo, sizeof (*mtuinfo));
6581     mtuinfo->ip6m_addr.sin6_family = AF_INET6;
6582     mtuinfo->ip6m_addr.sin6_port = connp->conn_fport;
6583     mtuinfo->ip6m_addr.sin6_addr = connp->conn_faddr_v6;
6584     mtuinfo->ip6m_addr.sin6_scopeid = scopeid;
6585     mtuinfo->ip6m_mtu = pmtu;
6586
6587     return (sizeof (struct ip6_mtuinfo));
6588 }
6589
6590 /*
6591  * When the src multihoming is changed from weak to [strong, preferred]
6592  * ip_ire_rebind_walker is called to walk the list of all ire_t entries
6593  * and identify routes that were created by user-applications in the
6594  * unbound state (i.e., without RTA_IFP), and for which an ire_ill is not
6595  * currently defined. These routes are then 'rebound', i.e., their ire_ill

```

```

6596 * is selected by finding an interface route for the gateway.
6597 */
6598 /* ARGSUSED */
6599 void
6600 ip_ire_rebind_walker(ire_t *ire, void *notused)
6601 {
6602     if (!ire->ire_unbound || ire->ire_ill != NULL)
6603         return;
6604     ire_rebind(ire);
6605     ire_delete(ire);
6606 }

6608 /*
6609  * When the src multihoming is changed from [strong, preferred] to weak,
6610  * ip_ire_unbind_walker is called to walk the list of all ire_t entries, and
6611  * set any entries that were created by user-applications in the unbound state
6612  * (i.e., without RTA_IFP) back to having a NULL ire_ill.
6613  */
6614 /* ARGSUSED */
6615 void
6616 ip_ire_unbind_walker(ire_t *ire, void *notused)
6617 {
6618     ire_t *new_ire;

6620     if (!ire->ire_unbound || ire->ire_ill == NULL)
6621         return;
6622     if (ire->ire_ipversion == IPV6_VERSION) {
6623         new_ire = ire_create_v6(&ire->ire_addr_v6, &ire->ire_mask_v6,
6624             &ire->ire_gateway_addr_v6, ire->ire_type, NULL,
6625             ire->ire_zoneid, ire->ire_flags, NULL, ire->ire_ipst);
6626     } else {
6627         new_ire = ire_create((uchar_t *)&ire->ire_addr,
6628             (uchar_t *)&ire->ire_mask,
6629             (uchar_t *)&ire->ire_gateway_addr, ire->ire_type, NULL,
6630             ire->ire_zoneid, ire->ire_flags, NULL, ire->ire_ipst);
6631     }
6632     if (new_ire == NULL)
6633         return;
6634     new_ire->ire_unbound = B_TRUE;
6635     /*
6636      * The bound ire must first be deleted so that we don't return
6637      * the existing one on the attempt to add the unbound new_ire.
6638      */
6639     ire_delete(ire);
6640     new_ire = ire_add(new_ire);
6641     if (new_ire != NULL)
6642         ire_refrele(new_ire);
6643 }

6645 /*
6646  * When the settings of ip*_strict_src_multihoming tunables are changed,
6647  * all cached routes need to be recomputed. This recomputation needs to be
6648  * done when going from weaker to stronger modes so that the cached ire
6649  * for the connection does not violate the current ip*_strict_src_multihoming
6650  * setting. It also needs to be done when going from stronger to weaker modes,
6651  * so that we fall back to matching on the longest-matching-route (as opposed
6652  * to a shorter match that may have been selected in the strong mode
6653  * to satisfy src_multihoming settings).
6654  */
6655 * The cached ixa_ire entires for all conn_t entries are marked as
6656 * "verify" so that they will be recomputed for the next packet.
6657 */
6658 void
6659 conn_ire_revalidate(conn_t *connp, void *arg)
6660 {
6661     boolean_t isv6 = (boolean_t)arg;

```

```

6663     if ((isv6 && connp->conn_ipversion != IPV6_VERSION) ||
6664         (!isv6 && connp->conn_ipversion != IPV4_VERSION))
6665         return;
6666     connp->conn_ixa->ixa_ire_generation = IRE_GENERATION_VERIFY;
6667 }

6669 /*
6670  * Handles both IPv4 and IPv6 reassembly - doing the out-of-order cases,
6671  * When an ipf is passed here for the first time, if
6672  * we already have in-order fragments on the queue, we convert from the fast-
6673  * path reassembly scheme to the hard-case scheme. From then on, additional
6674  * fragments are reassembled here. We keep track of the start and end offsets
6675  * of each piece, and the number of holes in the chain. When the hole count
6676  * goes to zero, we are done!
6677  */
6678 * The ipf_count will be updated to account for any mblk(s) added (pointed to
6679 * by mp) or subtracted (freeb(ed) dups), upon return the caller must update
6680 * ipfb_count and ill_frag_count by the difference of ipf_count before and
6681 * after the call to ip_reassemble().
6682 */
6683 int
6684 ip_reassemble(mblk_t *mp, ipf_t *ipf, uint_t start, boolean_t more, ill_t *ill,
6685     size_t msg_len)
6686 {
6687     uint_t end;
6688     mblk_t *next_mp;
6689     mblk_t *mpl;
6690     uint_t offset;
6691     boolean_t incr_dups = B_TRUE;
6692     boolean_t offset_zero_seen = B_FALSE;
6693     boolean_t pkt_boundary_checked = B_FALSE;

6695     /* If start == 0 then ipf_nf_hdr_len has to be set. */
6696     ASSERT(start != 0 || ipf->ipf_nf_hdr_len != 0);

6698     /* Add in byte count */
6699     ipf->ipf_count += msg_len;
6700     if (ipf->ipf_end) {
6701         /*
6702          * We were part way through in-order reassembly, but now there
6703          * is a hole. We walk through messages already queued, and
6704          * mark them for hard case reassembly. We know that up till
6705          * now they were in order starting from offset zero.
6706          */
6707         offset = 0;
6708         for (mpl = ipf->ipf_mp->b_cont; mpl; mpl = mpl->b_cont) {
6709             IP_REASS_SET_START(mpl, offset);
6710             if (offset == 0) {
6711                 ASSERT(ipf->ipf_nf_hdr_len != 0);
6712                 offset = -ipf->ipf_nf_hdr_len;
6713             }
6714             offset += mpl->b_wptr - mpl->b_rptr;
6715             IP_REASS_SET_END(mpl, offset);
6716         }
6717         /* One hole at the end. */
6718         ipf->ipf_hole_cnt = 1;
6719         /* Brand it as a hard case, forever. */
6720         ipf->ipf_end = 0;
6721     }
6722     /* Walk through all the new pieces. */
6723     do {
6724         end = start + (mp->b_wptr - mp->b_rptr);
6725         /*
6726          * If start is 0, decrease 'end' only for the first mblk of
6727          * the fragment. Otherwise 'end' can get wrong value in the

```

```

6728     * second pass of the loop if first mblk is exactly the
6729     * size of ipf_nf_hdr_len.
6730     */
6731     if (start == 0 && !offset_zero_seen) {
6732         /* First segment */
6733         ASSERT(ipf->ipf_nf_hdr_len != 0);
6734         end -= ipf->ipf_nf_hdr_len;
6735         offset_zero_seen = B_TRUE;
6736     }
6737     next_mp = mp->b_cont;
6738     /*
6739     * We are checking to see if there is any interesting data
6740     * to process. If there isn't and the mblk isn't the
6741     * one which carries the unfragmentable header then we
6742     * drop it. It's possible to have just the unfragmentable
6743     * header come through without any data. That needs to be
6744     * saved.
6745     *
6746     * If the assert at the top of this function holds then the
6747     * term "ipf->ipf_nf_hdr_len != 0" isn't needed. This code
6748     * is infrequently traveled enough that the test is left in
6749     * to protect against future code changes which break that
6750     * invariant.
6751     */
6752     if (start == end && start != 0 && ipf->ipf_nf_hdr_len != 0) {
6753         /* Empty. Blast it. */
6754         IP_REASS_SET_START(mp, 0);
6755         IP_REASS_SET_END(mp, 0);
6756         /*
6757         * If the ipf points to the mblk we are about to free,
6758         * update ipf to point to the next mblk (or NULL
6759         * if none).
6760         */
6761         if (ipf->ipf_mp->b_cont == mp)
6762             ipf->ipf_mp->b_cont = next_mp;
6763         freeb(mp);
6764         continue;
6765     }
6766     mp->b_cont = NULL;
6767     IP_REASS_SET_START(mp, start);
6768     IP_REASS_SET_END(mp, end);
6769     if (!ipf->ipf_tail_mp) {
6770         ipf->ipf_tail_mp = mp;
6771         ipf->ipf_mp->b_cont = mp;
6772         if (start == 0 || !more) {
6773             ipf->ipf_hole_cnt = 1;
6774             /*
6775             * if the first fragment comes in more than one
6776             * mblk, this loop will be executed for each
6777             * mblk. Need to adjust hole count so exiting
6778             * this routine will leave hole count at 1.
6779             */
6780             if (next_mp)
6781                 ipf->ipf_hole_cnt++;
6782         } else
6783             ipf->ipf_hole_cnt = 2;
6784         continue;
6785     } else if (ipf->ipf_last_frag_seen && !more &&
6786         !pkt_boundary_checked) {
6787         /*
6788         * We check datagram boundary only if this fragment
6789         * claims to be the last fragment and we have seen a
6790         * last fragment in the past too. We do this only
6791         * once for a given fragment.
6792         *
6793         * start cannot be 0 here as fragments with start=0

```

```

6794     * and MF=0 gets handled as a complete packet. These
6795     * fragments should not reach here.
6796     */
6797
6798     if (start + msgdsz(mp) !=
6799         IP_REASS_END(ipf->ipf_tail_mp)) {
6800         /*
6801         * We have two fragments both of which claim
6802         * to be the last fragment but gives conflicting
6803         * information about the whole datagram size.
6804         * Something fishy is going on. Drop the
6805         * fragment and free up the reassembly list.
6806         */
6807         return (IP_REASS_FAILED);
6808     }
6809
6810     /*
6811     * We shouldn't come to this code block again for this
6812     * particular fragment.
6813     */
6814     pkt_boundary_checked = B_TRUE;
6815 }
6816
6817 /* New stuff at or beyond tail? */
6818 offset = IP_REASS_END(ipf->ipf_tail_mp);
6819 if (start >= offset) {
6820     if (ipf->ipf_last_frag_seen) {
6821         /* current fragment is beyond last fragment */
6822         return (IP_REASS_FAILED);
6823     }
6824     /* Link it on end. */
6825     ipf->ipf_tail_mp->b_cont = mp;
6826     ipf->ipf_tail_mp = mp;
6827     if (more) {
6828         if (start != offset)
6829             ipf->ipf_hole_cnt++;
6830     } else if (start == offset && next_mp == NULL)
6831         ipf->ipf_hole_cnt--;
6832     continue;
6833 }
6834 mpl = ipf->ipf_mp->b_cont;
6835 offset = IP_REASS_START(mpl);
6836 /* New stuff at the front? */
6837 if (start < offset) {
6838     if (start == 0) {
6839         if (end >= offset) {
6840             /* Nailed the hole at the beginning. */
6841             ipf->ipf_hole_cnt--;
6842         }
6843     } else if (end < offset) {
6844         /*
6845         * A hole, stuff, and a hole where there used
6846         * to be just a hole.
6847         */
6848         ipf->ipf_hole_cnt++;
6849     }
6850     mp->b_cont = mpl;
6851     /* Check for overlap. */
6852     while (end > offset) {
6853         if (end < IP_REASS_END(mpl)) {
6854             mp->b_wptr -= end - offset;
6855             IP_REASS_SET_END(mp, offset);
6856             BUMP_MIB(ill->ill_ip_mib,
6857                 ipIfStatsReasmPartDups);
6858             break;
6859         }

```

```

6860      /* Did we cover another hole? */
6861      if ((mpl->b_cont &&
6862          IP_REASS_END(mpl) !=
6863          IP_REASS_START(mpl->b_cont) &&
6864          end >= IP_REASS_START(mpl->b_cont)) ||
6865          (!ipf->ipf_last_frag_seen && !more)) {
6866          ipf->ipf_hole_cnt--;
6867      }
6868      /* Clip out mpl. */
6869      if ((mp->b_cont = mpl->b_cont) == NULL) {
6870          /*
6871           * After clipping out mpl, this guy
6872           * is now hanging off the end.
6873           */
6874          ipf->ipf_tail_mp = mp;
6875      }
6876      IP_REASS_SET_START(mpl, 0);
6877      IP_REASS_SET_END(mpl, 0);
6878      /* Subtract byte count */
6879      ipf->ipf_count -= mpl->b_datap->db_lim -
6880          mpl->b_datap->db_base;
6881      freeb(mpl);
6882      BUMP_MIB(ill->ill_ip_mib,
6883          ipIfStatsReasmPartDups);
6884      mpl = mp->b_cont;
6885      if (!mpl)
6886          break;
6887      offset = IP_REASS_START(mpl);
6888  }
6889  ipf->ipf_mp->b_cont = mp;
6890  continue;
6891  }
6892  /*
6893   * The new piece starts somewhere between the start of the head
6894   * and before the end of the tail.
6895   */
6896  for (; mpl; mpl = mpl->b_cont) {
6897      offset = IP_REASS_END(mpl);
6898      if (start < offset) {
6899          if (end <= offset) {
6900              /* Nothing new. */
6901              IP_REASS_SET_START(mp, 0);
6902              IP_REASS_SET_END(mp, 0);
6903              /* Subtract byte count */
6904              ipf->ipf_count -= mp->b_datap->db_lim -
6905                  mp->b_datap->db_base;
6906              if (incr_dups) {
6907                  ipf->ipf_num_dups++;
6908                  incr_dups = B_FALSE;
6909              }
6910              freeb(mp);
6911              BUMP_MIB(ill->ill_ip_mib,
6912                  ipIfStatsReasmDuplicates);
6913              break;
6914          }
6915          /*
6916           * Trim redundant stuff off beginning of new
6917           * piece.
6918           */
6919          IP_REASS_SET_START(mp, offset);
6920          mp->b_rptr += offset - start;
6921          BUMP_MIB(ill->ill_ip_mib,
6922              ipIfStatsReasmPartDups);
6923          start = offset;
6924          if (!mpl->b_cont) {
6925              /*

```

```

6926          * After trimming, this guy is now
6927          * hanging off the end.
6928          */
6929          mpl->b_cont = mp;
6930          ipf->ipf_tail_mp = mp;
6931          if (!more) {
6932              ipf->ipf_hole_cnt--;
6933          }
6934          break;
6935      }
6936  }
6937  if (start >= IP_REASS_START(mpl->b_cont))
6938      continue;
6939  /* Fill a hole */
6940  if (start > offset)
6941      ipf->ipf_hole_cnt++;
6942  mp->b_cont = mpl->b_cont;
6943  mpl->b_cont = mp;
6944  mpl = mp->b_cont;
6945  offset = IP_REASS_START(mpl);
6946  if (end >= offset) {
6947      ipf->ipf_hole_cnt--;
6948      /* Check for overlap. */
6949      while (end > offset) {
6950          if (end < IP_REASS_END(mpl)) {
6951              mp->b_wptr -= end - offset;
6952              IP_REASS_SET_END(mp, offset);
6953              /*
6954               * TODO we might bump
6955               * this up twice if there is
6956               * overlap at both ends.
6957               */
6958              BUMP_MIB(ill->ill_ip_mib,
6959                  ipIfStatsReasmPartDups);
6960              break;
6961          }
6962      }
6963      /* Did we cover another hole? */
6964      if ((mpl->b_cont &&
6965          IP_REASS_END(mpl)
6966          != IP_REASS_START(mpl->b_cont) &&
6967          end >=
6968          IP_REASS_START(mpl->b_cont)) ||
6969          (!ipf->ipf_last_frag_seen &&
6970          !more)) {
6971          ipf->ipf_hole_cnt--;
6972      }
6973      /* Clip out mpl. */
6974      if ((mp->b_cont = mpl->b_cont) ==
6975          NULL) {
6976          /*
6977           * After clipping out mpl,
6978           * this guy is now hanging
6979           * off the end.
6980           */
6981          ipf->ipf_tail_mp = mp;
6982      }
6983      IP_REASS_SET_START(mpl, 0);
6984      IP_REASS_SET_END(mpl, 0);
6985      /* Subtract byte count */
6986      ipf->ipf_count -=
6987          mpl->b_datap->db_lim -
6988          mpl->b_datap->db_base;
6989      freeb(mpl);
6990      BUMP_MIB(ill->ill_ip_mib,
6991          ipIfStatsReasmPartDups);
6992      mpl = mp->b_cont;

```

```

6992         if (!mpl)
6993             break;
6994         offset = IP_REASS_START(mpl);
6995     }
6996     }
6997     break;
6998 }
6999 } while (start = end, mp = next_mp);

7001 /* Fragment just processed could be the last one. Remember this fact */
7002 if (!more)
7003     ipf->ipf_last_frag_seen = B_TRUE;

7005 /* Still got holes? */
7006 if (ipf->ipf_hole_cnt)
7007     return (IP_REASS_PARTIAL);
7008 /* Clean up overloaded fields to avoid upstream disasters. */
7009 for (mpl = ipf->ipf_mp->b_cont; mpl; mpl = mpl->b_cont) {
7010     IP_REASS_SET_START(mpl, 0);
7011     IP_REASS_SET_END(mpl, 0);
7012 }
7013 return (IP_REASS_COMPLETE);
7014 }

7016 /*
7017  * Fragmentation reassembly. Each ILL has a hash table for
7018  * queuing packets undergoing reassembly for all IPIFs
7019  * associated with the ILL. The hash is based on the packet
7020  * IP ident field. The ILL frag hash table was allocated
7021  * as a timer block at the time the ILL was created. Whenever
7022  * there is anything on the reassembly queue, the timer will
7023  * be running. Returns the reassembled packet if reassembly completes.
7024  */
7025 mblk_t *
7026 ip_input_fragment(mblk_t *mp, ipha_t *ipha, ip_rcv_attr_t *ira)
7027 {
7028     uint32_t    frag_offset_flags;
7029     mblk_t      *t_mp;
7030     ipaddr_t    dst;
7031     uint8_t      proto = ipha->ipha_protocol;
7032     uint32_t     sum_val;
7033     uint16_t     sum_flags;
7034     ipf_t        *ipf;
7035     ipf_t        **ipfp;
7036     ipfb_t       *ipfb;
7037     uint16_t     ident;
7038     uint32_t     offset;
7039     ipaddr_t     src;
7040     uint_t       hdr_length;
7041     uint32_t     end;
7042     mblk_t       *mpl;
7043     mblk_t       *tail_mp;
7044     size_t       count;
7045     size_t       msg_len;
7046     uint8_t      ecn_info = 0;
7047     uint32_t     packet_size;
7048     boolean_t     pruned = B_FALSE;
7049     ill_t        *ill = ira->ira_ill;
7050     ip_stack_t   *ipst = ill->ill_ipst;

7052 /*
7053  * Drop the fragmented as early as possible, if
7054  * we don't have resource(s) to re-assemble.
7055  */
7056 if (ipst->ips_ip_reass_queue_bytes == 0) {
7057     freemsg(mp);

```

```

7058         return (NULL);
7059     }

7061 /* Check for fragmentation offset; return if there's none */
7062 if ((frag_offset_flags = ntohs(ipha->ipha_fragment_offset_and_flags) &
7063     (IPH_MF | IPH_OFFSET)) == 0)
7064     return (mp);

7066 /*
7067  * We utilize hardware computed checksum info only for UDP since
7068  * IP fragmentation is a normal occurrence for the protocol. In
7069  * addition, checksum offload support for IP fragments carrying
7070  * UDP payload is commonly implemented across network adapters.
7071  */
7072 ASSERT(ira->ira_rill != NULL);
7073 if (proto == IPPROTO_UDP && dohwcksum &&
7074     ILL_HCKSUM_CAPABLE(ira->ira_rill) &&
7075     (DB_CKSUMFLAGS(mp) & (HCK_FULLCKSUM | HCK_PARTIALCKSUM))) {
7076     mblk_t *mpl = mp->b_cont;
7077     int32_t len;

7079     /* Record checksum information from the packet */
7080     sum_val = (uint32_t)DB_CKSUM16(mp);
7081     sum_flags = DB_CKSUMFLAGS(mp);

7083     /* IP payload offset from beginning of mblk */
7084     offset = ((uchar_t *)ipha + IPH_HDR_LENGTH(ipha)) - mp->b_rptr;

7086     if ((sum_flags & HCK_PARTIALCKSUM) &&
7087         (mpl == NULL || mpl->b_cont == NULL) &&
7088         offset >= DB_CKSUMSTART(mp) &&
7089         ((len = offset - DB_CKSUMSTART(mp)) & 1) == 0) {
7090         uint32_t adj;
7091         /*
7092          * Partial checksum has been calculated by hardware
7093          * and attached to the packet; in addition, any
7094          * prepended extraneous data is even byte aligned.
7095          * If any such data exists, we adjust the checksum;
7096          * this would also handle any postpend data.
7097          */
7098         IP_ADJCKSUM_PARTIAL(mp->b_rptr + DB_CKSUMSTART(mp),
7099             mp, mpl, len, adj);

7101         /* One's complement subtract extraneous checksum */
7102         if (adj >= sum_val)
7103             sum_val = ~(adj - sum_val) & 0xFFFF;
7104         else
7105             sum_val -= adj;
7106     }
7107 } else {
7108     sum_val = 0;
7109     sum_flags = 0;
7110 }

7112 /* Clear hardware checksumming flag */
7113 DB_CKSUMFLAGS(mp) = 0;

7115 ident = ipha->ipha_ident;
7116 offset = (frag_offset_flags << 3) & 0xFFFF;
7117 src = ipha->ipha_src;
7118 dst = ipha->ipha_dst;
7119 hdr_length = IPH_HDR_LENGTH(ipha);
7120 end = ntohs(ipha->ipha_length) - hdr_length;

7122 /* If end == 0 then we have a packet with no data, so just free it */
7123 if (end == 0) {

```

```

7124         freemsg(mp);
7125         return (NULL);
7126     }

7128     /* Record the ECN field info. */
7129     ecn_info = (ipha->ipha_type_of_service & 0x3);
7130     if (offset != 0) {
7131         /*
7132          * If this isn't the first piece, strip the header, and
7133          * add the offset to the end value.
7134          */
7135         mp->b_rptr += hdr_length;
7136         end += offset;
7137     }

7139     /* Handle vnic loopback of fragments */
7140     if (mp->b_datap->db_ref > 2)
7141         msg_len = 0;
7142     else
7143         msg_len = MBLKSIZE(mp);

7145     tail_mp = mp;
7146     while (tail_mp->b_cont != NULL) {
7147         tail_mp = tail_mp->b_cont;
7148         if (tail_mp->b_datap->db_ref <= 2)
7149             msg_len += MBLKSIZE(tail_mp);
7150     }

7152     /* If the reassembly list for this ILL will get too big, prune it */
7153     if ((msg_len + sizeof (*ipf) + ill->ill_frag_count) >=
7154         ipst->ips_ip_reass_queue_bytes) {
7155         DTRACE_PROBE3(ip_reass_queue_bytes, uint_t, msg_len,
7156             uint_t, ill->ill_frag_count,
7157             uint_t, ipst->ips_ip_reass_queue_bytes);
7158         ill_frag_prune(ill,
7159             (ipst->ips_ip_reass_queue_bytes < msg_len) ? 0 :
7160             (ipst->ips_ip_reass_queue_bytes - msg_len));
7161         pruned = B_TRUE;
7162     }

7164     ipfb = &ill->ill_frag_hash_tbl[ILL_FRAG_HASH(src, ident)];
7165     mutex_enter(&ipfb->ipfb_lock);

7167     ipfp = &ipfb->ipfb_ipf;
7168     /* Try to find an existing fragment queue for this packet. */
7169     for (;;) {
7170         ipf = ipfp[0];
7171         if (ipf != NULL) {
7172             /*
7173              * It has to match on ident and src/dst address.
7174              */
7175             if (ipf->ipf_ident == ident &&
7176                 ipf->ipf_src == src &&
7177                 ipf->ipf_dst == dst &&
7178                 ipf->ipf_protocol == proto) {
7179                 /*
7180                  * If we have received too many
7181                  * duplicate fragments for this packet
7182                  * free it.
7183                  */
7184                 if (ipf->ipf_num_dups > ip_max_frag_dups) {
7185                     ill_frag_free_pkts(ill, ipfb, ipf, 1);
7186                     freemsg(mp);
7187                     mutex_exit(&ipfb->ipfb_lock);
7188                     return (NULL);
7189                 }

```

```

7190             /* Found it. */
7191             break;
7192         }
7193         ipfp = &ipfb->ipfb_hash_next;
7194         continue;
7195     }

7197     /*
7198     * If we pruned the list, do we want to store this new
7199     * fragment?. We apply an optimization here based on the
7200     * fact that most fragments will be received in order.
7201     * So if the offset of this incoming fragment is zero,
7202     * it is the first fragment of a new packet. We will
7203     * keep it. Otherwise drop the fragment, as we have
7204     * probably pruned the packet already (since the
7205     * packet cannot be found).
7206     */
7207     if (pruned && offset != 0) {
7208         mutex_exit(&ipfb->ipfb_lock);
7209         freemsg(mp);
7210         return (NULL);
7211     }

7213     if (ipfb->ipfb_frag_pkts >= MAX_FRAG_PKTS(ipst)) {
7214         /*
7215          * Too many fragmented packets in this hash
7216          * bucket. Free the oldest.
7217          */
7218         ill_frag_free_pkts(ill, ipfb, ipfb->ipfb_ipf, 1);
7219     }

7221     /* New guy. Allocate a frag message. */
7222     mp1 = allocb(sizeof (*ipf), BPRI_MED);
7223     if (mp1 == NULL) {
7224         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7225         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7226         freemsg(mp);
7227     reass_done:
7228         mutex_exit(&ipfb->ipfb_lock);
7229         return (NULL);
7230     }

7232     BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmReqds);
7233     mp1->b_cont = mp;

7235     /* Initialize the fragment header. */
7236     ipf = (ipf_t *)mp1->b_rptr;
7237     ipf->ipf_mp = mp1;
7238     ipf->ipf_ptphn = ipfp;
7239     ipfp[0] = ipf;
7240     ipf->ipf_hash_next = NULL;
7241     ipf->ipf_ident = ident;
7242     ipf->ipf_protocol = proto;
7243     ipf->ipf_src = src;
7244     ipf->ipf_dst = dst;
7245     ipf->ipf_nf_hdr_len = 0;
7246     /* Record reassembly start time. */
7247     ipf->ipf_timestamp = gethrestime_sec();
7248     /* Record ipf generation and account for frag header */
7249     ipf->ipf_gen = ill->ill_ipf_gen++;
7250     ipf->ipf_count = MBLKSIZE(mp1);
7251     ipf->ipf_last_frag_seen = B_FALSE;
7252     ipf->ipf_ecn = ecn_info;
7253     ipf->ipf_num_dups = 0;
7254     ipfb->ipfb_frag_pkts++;
7255     ipf->ipf_checksum = 0;

```



```

7256         ipf->ipf_checksum_flags = 0;

7258         /* Store checksum value in fragment header */
7259         if (sum_flags != 0) {
7260             sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7261             sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7262             ipf->ipf_checksum = sum_val;
7263             ipf->ipf_checksum_flags = sum_flags;
7264         }

7266         /*
7267          * We handle reassembly two ways. In the easy case,
7268          * where all the fragments show up in order, we do
7269          * minimal bookkeeping, and just clip new pieces on
7270          * the end. If we ever see a hole, then we go off
7271          * to ip_reassemble which has to mark the pieces and
7272          * keep track of the number of holes, etc. Obviously,
7273          * the point of having both mechanisms is so we can
7274          * handle the easy case as efficiently as possible.
7275          */
7276         if (offset == 0) {
7277             /* Easy case, in-order reassembly so far. */
7278             ipf->ipf_count += msg_len;
7279             ipf->ipf_tail_mp = tail_mp;
7280             /*
7281              * Keep track of next expected offset in
7282              * ipf_end.
7283              */
7284             ipf->ipf_end = end;
7285             ipf->ipf_nf_hdr_len = hdr_length;
7286         } else {
7287             /* Hard case, hole at the beginning. */
7288             ipf->ipf_tail_mp = NULL;
7289             /*
7290              * ipf_end == 0 means that we have given up
7291              * on easy reassembly.
7292              */
7293             ipf->ipf_end = 0;

7295             /* Forget checksum offload from now on */
7296             ipf->ipf_checksum_flags = 0;

7298             /*
7299              * ipf_hole_cnt is set by ip_reassemble.
7300              * ipf_count is updated by ip_reassemble.
7301              * No need to check for return value here
7302              * as we don't expect reassembly to complete
7303              * or fail for the first fragment itself.
7304              */
7305             (void) ip_reassemble(mp, ipf,
7306                 (frag_offset_flags & IPH_OFFSET) << 3,
7307                 (frag_offset_flags & IPH_MF), ill, msg_len);
7308         }
7309         /* Update per ipfb and ill byte counts */
7310         ipfb->ipfb_count += ipf->ipf_count;
7311         ASSERT(ipfb->ipfb_count > 0); /* Wraparound */
7312         atomic_add_32(&ill->ill_frag_count, ipf->ipf_count);
7313         /* If the frag timer wasn't already going, start it. */
7314         mutex_enter(&ill->ill_lock);
7315         ill_frag_timer_start(ill);
7316         mutex_exit(&ill->ill_lock);
7317         goto reass_done;
7318     }

7320     /*
7321      * If the packet's flag has changed (it could be coming up

```

```

7322         * from an interface different than the previous, therefore
7323         * possibly different checksum capability), then forget about
7324         * any stored checksum states. Otherwise add the value to
7325         * the existing one stored in the fragment header.
7326         */
7327         if (sum_flags != 0 && sum_flags == ipf->ipf_checksum_flags) {
7328             sum_val += ipf->ipf_checksum;
7329             sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7330             sum_val = (sum_val & 0xFFFF) + (sum_val >> 16);
7331             ipf->ipf_checksum = sum_val;
7332         } else if (ipf->ipf_checksum_flags != 0) {
7333             /* Forget checksum offload from now on */
7334             ipf->ipf_checksum_flags = 0;
7335         }

7337         /*
7338          * We have a new piece of a datagram which is already being
7339          * reassembled. Update the ECN info if all IP fragments
7340          * are ECN capable. If there is one which is not, clear
7341          * all the info. If there is at least one which has CE
7342          * code point, IP needs to report that up to transport.
7343          */
7344         if (ecn_info != IPH_ECN_NECT && ipf->ipf_ecn != IPH_ECN_NECT) {
7345             if (ecn_info == IPH_ECN_CE)
7346                 ipf->ipf_ecn = IPH_ECN_CE;
7347         } else {
7348             ipf->ipf_ecn = IPH_ECN_NECT;
7349         }
7350         if (offset && ipf->ipf_end == offset) {
7351             /* The new fragment fits at the end */
7352             ipf->ipf_tail_mp->b_cont = mp;
7353             /* Update the byte count */
7354             ipf->ipf_count += msg_len;
7355             /* Update per ipfb and ill byte counts */
7356             ipfb->ipfb_count += msg_len;
7357             ASSERT(ipfb->ipfb_count > 0); /* Wraparound */
7358             atomic_add_32(&ill->ill_frag_count, msg_len);
7359             if (frag_offset_flags & IPH_MF) {
7360                 /* More to come. */
7361                 ipf->ipf_end = end;
7362                 ipf->ipf_tail_mp = tail_mp;
7363                 goto reass_done;
7364             }
7365         } else {
7366             /* Go do the hard cases. */
7367             int ret;

7369             if (offset == 0)
7370                 ipf->ipf_nf_hdr_len = hdr_length;

7372             /* Save current byte count */
7373             count = ipf->ipf_count;
7374             ret = ip_reassemble(mp, ipf,
7375                 (frag_offset_flags & IPH_OFFSET) << 3,
7376                 (frag_offset_flags & IPH_MF), ill, msg_len);
7377             /* Count of bytes added and subtracted (freeb()) */
7378             count = ipf->ipf_count - count;
7379             if (count) {
7380                 /* Update per ipfb and ill byte counts */
7381                 ipfb->ipfb_count += count;
7382                 ASSERT(ipfb->ipfb_count > 0); /* Wraparound */
7383                 atomic_add_32(&ill->ill_frag_count, count);
7384             }
7385             if (ret == IP_REASS_PARTIAL) {
7386                 goto reass_done;
7387             } else if (ret == IP_REASS_FAILED) {

```

```

7388      /* Reassembly failed. Free up all resources */
7389      ill_frag_free_pkts(ill, ipfb, ipf, 1);
7390      for (t_mp = mp; t_mp != NULL; t_mp = t_mp->b_cont) {
7391          IP_REASS_SET_START(t_mp, 0);
7392          IP_REASS_SET_END(t_mp, 0);
7393      }
7394      freemsg(mp);
7395      goto reass_done;
7396  }
7397  /* We will reach here iff 'ret' is IP_REASS_COMPLETE */
7398  }
7399  /*
7400   * We have completed reassembly. Unhook the frag header from
7401   * the reassembly list.
7402   *
7403   * Before we free the frag header, record the ECN info
7404   * to report back to the transport.
7405   */
7406  ecn_info = ipf->ipf_ecn;
7407  BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmOKs);
7408  ipfp = ipf->ipf_ptphn;

7410  /* We need to supply these to caller */
7411  if ((sum_flags = ipf->ipf_checksum_flags) != 0)
7412      sum_val = ipf->ipf_checksum;
7413  else
7414      sum_val = 0;

7416  mpl = ipf->ipf_mp;
7417  count = ipf->ipf_count;
7418  ipf = ipf->ipf_hash_next;
7419  if (ipf != NULL)
7420      ipf->ipf_ptphn = ipfp;
7421  ipfp[0] = ipf;
7422  atomic_add_32(&ill->ill_frag_count, -count);
7423  ASSERT(ipfb->ipfb_count >= count);
7424  ipfb->ipfb_count -= count;
7425  ipfb->ipfb_frag_pkts--;
7426  mutex_exit(&ipfb->ipfb_lock);
7427  /* Ditch the frag header. */
7428  mp = mpl->b_cont;

7430  freeb(mpl);

7432  /* Restore original IP length in header. */
7433  packet_size = (uint32_t)msgdsize(mp);
7434  if (packet_size > IP_MAXPACKET) {
7435      BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7436      ip_drop_input("Reassembled packet too large", mp, ill);
7437      freemsg(mp);
7438      return (NULL);
7439  }

7441  if (DB_REF(mp) > 1) {
7442      mblk_t *mp2 = copymsg(mp);

7444      if (mp2 == NULL) {
7445          BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7446          ip_drop_input("ipIfStatsInDiscards", mp, ill);
7447          freemsg(mp);
7448          return (NULL);
7449      }
7450      freemsg(mp);
7451      mp = mp2;
7452  }
7453  ipha = (ipha_t *)mp->b_rptr;

```

```

7455      ipha->ipha_length = htons((uint16_t)packet_size);
7456      /* We're now complete, zip the frag state */
7457      ipha->ipha_fragment_offset_and_flags = 0;
7458      /* Record the ECN info. */
7459      ipha->ipha_type_of_service &= 0xFC;
7460      ipha->ipha_type_of_service |= ecn_info;

7462      /* Update the receive attributes */
7463      ira->ira_pktlen = packet_size;
7464      ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);

7466      /* Reassembly is successful; set checksum information in packet */
7467      DB_CKSUM16(mp) = (uint16_t)sum_val;
7468      DB_CKSUMFLAGS(mp) = sum_flags;
7469      DB_CKSUMSTART(mp) = ira->ira_ip_hdr_length;

7471      return (mp);
7472  }

7474  /*
7475   * Pullup function that should be used for IP input in order to
7476   * ensure we do not loose the L2 source address; we need the l2 source
7477   * address for IP_RECVSLLA and for ndp_input.
7478   *
7479   * We return either NULL or b_rptr.
7480   */
7481  void *
7482  ip_pullup(mblk_t *mp, ssize_t len, ip_recv_attr_t *ira)
7483  {
7484      ill_t      *ill = ira->ira_ill;

7486      if (ip_rput_pullups++ == 0) {
7487          (void) mi_strlog(ill->ill_rq, 1, SL_ERROR|SL_TRACE,
7488              "ip_pullup: %s forced us to ",
7489              " pullup pkt, hdr len %ld, hdr addr %p",
7490              ill->ill_name, len, (void *)mp->b_rptr);
7491      }
7492      if (!(ira->ira_flags & IRAF_L2SRC_SET))
7493          ip_setl2src(mp, ira, ira->ira_rill);
7494      ASSERT(ira->ira_flags & IRAF_L2SRC_SET);
7495      if (!pullupmsg(mp, len))
7496          return (NULL);
7497      else
7498          return (mp->b_rptr);
7499  }

7501  /*
7502   * Make sure ira_l2src has an address. If we don't have one fill with zeros.
7503   * When called from the ULP ira_rill will be NULL hence the caller has to
7504   * pass in the ill.
7505   */
7506  /* ARGSUSED */
7507  void
7508  ip_setl2src(mblk_t *mp, ip_recv_attr_t *ira, ill_t *ill)
7509  {
7510      const uchar_t *addr;
7511      int alen;

7513      if (ira->ira_flags & IRAF_L2SRC_SET)
7514          return;

7516      ASSERT(ill != NULL);
7517      alen = ill->ill_phys_addr_length;
7518      ASSERT(alen <= sizeof (ira->ira_l2src));
7519      if (ira->ira_mhip != NULL &&

```

```

7520     (addr = ira->ira_mhip->mhi_saddr) != NULL) {
7521         bcopy(addr, ira->ira_l2src, alen);
7522     } else if ((ira->ira_flags & IRAF_L2SRC_LOOPBACK) &&
7523         (addr = ill->ill_phys_addr) != NULL) {
7524         bcopy(addr, ira->ira_l2src, alen);
7525     } else {
7526         bzero(ira->ira_l2src, alen);
7527     }
7528     ira->ira_flags |= IRAF_L2SRC_SET;
7529 }

7531 /*
7532  * check ip header length and align it.
7533  */
7534 mblk_t *
7535 ip_check_and_align_header(mblk_t *mp, uint_t min_size, ip_rcv_attr_t *ira)
7536 {
7537     ill_t *ill = ira->ira_ill;
7538     ssize_t len;

7540     len = MBLKL(mp);

7542     if (!OK_32PTR(mp->b_rptr))
7543         IP_STAT(ill->ill_ipst, ip_notaligned);
7544     else
7545         IP_STAT(ill->ill_ipst, ip_rcv_pullup);

7547     /* Guard against bogus device drivers */
7548     if (len < 0) {
7549         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7550         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7551         freemsg(mp);
7552         return (NULL);
7553     }

7555     if (len == 0) {
7556         /* GLD sometimes sends up mblk with b_rptr == b_wptr! */
7557         mblk_t *mpl = mp->b_cont;

7559         if (!(ira->ira_flags & IRAF_L2SRC_SET))
7560             ip_setl2src(mp, ira, ira->ira_rill);
7561         ASSERT(ira->ira_flags & IRAF_L2SRC_SET);

7563         freeb(mp);
7564         mp = mpl;
7565         if (mp == NULL)
7566             return (NULL);

7568         if (OK_32PTR(mp->b_rptr) && MBLKL(mp) >= min_size)
7569             return (mp);
7570     }
7571     if (ip_pullup(mp, min_size, ira) == NULL) {
7572         if (msgdsize(mp) < min_size) {
7573             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7574             ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7575         } else {
7576             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7577             ip_drop_input("ipIfStatsInDiscards", mp, ill);
7578         }
7579         freemsg(mp);
7580         return (NULL);
7581     }
7582     return (mp);
7583 }

7585 /*

```

```

7586  * Common code for IPv4 and IPv6 to check and pullup multi-mblks
7587  */
7588 mblk_t *
7589 ip_check_length(mblk_t *mp, uchar_t *rptr, ssize_t len, uint_t pkt_len,
7590     uint_t min_size, ip_rcv_attr_t *ira)
7591 {
7592     ill_t *ill = ira->ira_ill;

7594     /*
7595      * Make sure we have data length consistent
7596      * with the IP header.
7597      */
7598     if (mp->b_cont == NULL) {
7599         /* pkt_len is based on ipha_len, not the mblk length */
7600         if (pkt_len < min_size) {
7601             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7602             ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7603             freemsg(mp);
7604             return (NULL);
7605         }
7606         if (len < 0) {
7607             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
7608             ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
7609             freemsg(mp);
7610             return (NULL);
7611         }
7612         /* Drop any pad */
7613         mp->b_wptr = rptr + pkt_len;
7614     } else if ((len += msgdsize(mp->b_cont)) != 0) {
7615         ASSERT(pkt_len >= min_size);
7616         if (pkt_len < min_size) {
7617             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7618             ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7619             freemsg(mp);
7620             return (NULL);
7621         }
7622         if (len < 0) {
7623             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
7624             ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
7625             freemsg(mp);
7626             return (NULL);
7627         }
7628         /* Drop any pad */
7629         (void) adjmsg(mp, -len);
7630         /*
7631          * adjmsg may have freed an mblk from the chain, hence
7632          * invalidate any hw checksum here. This will force IP to
7633          * calculate the checksum in sw, but only for this packet.
7634          */
7635         DB_CKSUMFLAGS(mp) = 0;
7636         IP_STAT(ill->ill_ipst, ip_multimblk);
7637     }
7638     return (mp);
7639 }

7641 /*
7642  * Check that the IPv4 opt_len is consistent with the packet and pullup
7643  * the options.
7644  */
7645 mblk_t *
7646 ip_check_optlen(mblk_t *mp, ipha_t *ipha, uint_t opt_len, uint_t pkt_len,
7647     ip_rcv_attr_t *ira)
7648 {
7649     ill_t *ill = ira->ira_ill;
7650     ssize_t len;

```

```

7652 /* Assume no IPv6 packets arrive over the IPv4 queue */
7653 if (IPH_HDR_VERSION(ipha) != IPV4_VERSION) {
7654     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7655     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInWrongIPVersion);
7656     ip_drop_input("IPvN packet on IPv4 ill", mp, ill);
7657     freemsg(mp);
7658     return (NULL);
7659 }
7660
7661 if (opt_len > (15 - IP_SIMPLE_HDR_LENGTH_IN_WORDS)) {
7662     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7663     ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7664     freemsg(mp);
7665     return (NULL);
7666 }
7667 /*
7668  * Recompute complete header length and make sure we
7669  * have access to all of it.
7670  */
7671 len = ((size_t)opt_len + IP_SIMPLE_HDR_LENGTH_IN_WORDS) << 2;
7672 if (len > (mp->b_wptr - mp->b_rptr)) {
7673     if (len > pkt_len) {
7674         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInHdrErrors);
7675         ip_drop_input("ipIfStatsInHdrErrors", mp, ill);
7676         freemsg(mp);
7677         return (NULL);
7678     }
7679     if (ip_pullup(mp, len, ira) == NULL) {
7680         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
7681         ip_drop_input("ipIfStatsInDiscards", mp, ill);
7682         freemsg(mp);
7683         return (NULL);
7684     }
7685 }
7686 return (mp);
7687 }
7688
7689 /*
7690  * Returns a new ire, or the same ire, or NULL.
7691  * If a different IRE is returned, then it is held; the caller
7692  * needs to release it.
7693  * In no case is there any hold/release on the ire argument.
7694  */
7695 ire_t *
7696 ip_check_multihome(void *addr, ire_t *ire, ill_t *ill)
7697 {
7698     ire_t      *new_ire;
7699     ill_t      *ire_ill;
7700     uint_t     ifindex;
7701     ip_stack_t *ipst = ill->ill_ipst;
7702     boolean_t   strict_check = B_FALSE;
7703
7704     /*
7705      * IPMP common case: if IRE and ILL are in the same group, there's no
7706      * issue (e.g. packet received on an underlying interface matched an
7707      * IRE_LOCAL on its associated group interface).
7708      */
7709     ASSERT(ire->ire_ill != NULL);
7710     if (IS_IN_SAME_ILLGRP(ill, ire->ire_ill))
7711         return (ire);
7712
7713     /*
7714      * Do another ire lookup here, using the ingress ill, to see if the
7715      * interface is in a usesrc group.
7716      * As long as the ills belong to the same group, we don't consider
7717      * them to be arriving on the wrong interface. Thus, if the switch

```

```

7718     * is doing inbound load spreading, we won't drop packets when the
7719     * ip*_strict_dst_multihoming switch is on.
7720     * We also need to check for IPIF_UNNUMBERED point2point interfaces
7721     * where the local address may not be unique. In this case we were
7722     * at the mercy of the initial ire lookup and the IRE_LOCAL it
7723     * actually returned. The new lookup, which is more specific, should
7724     * only find the IRE_LOCAL associated with the ingress ill if one
7725     * exists.
7726     */
7727     if (ire->ire_ipversion == IPV4_VERSION) {
7728         if (ipst->ips_ip_strict_dst_multihoming)
7729             strict_check = B_TRUE;
7730         new_ire = ire_ftable_lookup_v4(((ipaddr_t *)addr), 0, 0,
7731             IRE_LOCAL, ill, ALL_ZONES, NULL,
7732             (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
7733     } else {
7734         ASSERT(!IN6_IS_ADDR_MULTICAST((in6_addr_t *)addr));
7735         if (ipst->ips_ipv6_strict_dst_multihoming)
7736             strict_check = B_TRUE;
7737         new_ire = ire_ftable_lookup_v6((in6_addr_t *)addr, NULL, NULL,
7738             IRE_LOCAL, ill, ALL_ZONES, NULL,
7739             (MATCH_IRE_TYPE|MATCH_IRE_ILL), 0, ipst, NULL);
7740     }
7741     /*
7742      * If the same ire that was returned in ip_input() is found then this
7743      * is an indication that usesrc groups are in use. The packet
7744      * arrived on a different ill in the group than the one associated with
7745      * the destination address. If a different ire was found then the same
7746      * IP address must be hosted on multiple ills. This is possible with
7747      * unnumbered point2point interfaces. We switch to use this new ire in
7748      * order to have accurate interface statistics.
7749     */
7750     if (new_ire != NULL) {
7751         /* Note: held in one case but not the other? Caller handles */
7752         if (new_ire != ire)
7753             return (new_ire);
7754         /* Unchanged */
7755         ire_refrele(new_ire);
7756         return (ire);
7757     }
7758
7759     /*
7760      * Chase pointers once and store locally.
7761      */
7762     ASSERT(ire->ire_ill != NULL);
7763     ire_ill = ire->ire_ill;
7764     ifindex = ill->ill_usesrc_ifindex;
7765
7766     /*
7767      * Check if it's a legal address on the 'usesrc' interface.
7768      * For IPMP data addresses the IRE_LOCAL is the upper, hence we
7769      * can just check phyint_ifindex.
7770     */
7771     if (ifindex != 0 && ifindex == ire_ill->ill_phyint->phyint_ifindex) {
7772         return (ire);
7773     }
7774
7775     /*
7776      * If the ip*_strict_dst_multihoming switch is on then we can
7777      * only accept this packet if the interface is marked as routing.
7778     */
7779     if (!(strict_check))
7780         return (ire);
7781
7782     if ((ill->ill_flags & ire->ire_ill->ill_flags & ILLF_ROUTER) != 0) {
7783         return (ire);

```

```

7784     }
7785     return (NULL);
7786 }

7788 /*
7789  * This function is used to construct a mac_header_info_s from a
7790  * DL_UNITDATA_IND message.
7791  * The address fields in the mhi structure points into the message,
7792  * thus the caller can't use those fields after freeing the message.
7793  */
7794 * We determine whether the packet received is a non-unicast packet
7795 * and in doing so, determine whether or not it is broadcast vs multicast.
7796 * For it to be a broadcast packet, we must have the appropriate mblk_t
7797 * hanging off the ill_t. If this is either not present or doesn't match
7798 * the destination mac address in the DL_UNITDATA_IND, the packet is deemed
7799 * to be multicast. Thus NICs that have no broadcast address (or no
7800 * capability for one, such as point to point links) cannot return as
7801 * the packet being broadcast.
7802 */
7803 void
7804 ip_dlur_to_mhi(ill_t *ill, mblk_t *mb, struct mac_header_info_s *mhip)
7805 {
7806     dl_unitdata_ind_t *ind = (dl_unitdata_ind_t *)mb->b_rptr;
7807     mblk_t *bmp;
7808     uint_t extra_offset;

7810     bzero(mhip, sizeof (struct mac_header_info_s));

7812     mhip->mhi_dsttype = MAC_ADDRTYPE_UNICAST;

7814     if (ill->ill_sap_length < 0)
7815         extra_offset = 0;
7816     else
7817         extra_offset = ill->ill_sap_length;

7819     mhip->mhi_daddr = (uchar_t *)ind + ind->dl_dest_addr_offset +
7820         extra_offset;
7821     mhip->mhi_saddr = (uchar_t *)ind + ind->dl_src_addr_offset +
7822         extra_offset;

7824     if (!ind->dl_group_address)
7825         return;

7827     /* Multicast or broadcast */
7828     mhip->mhi_dsttype = MAC_ADDRTYPE_MULTICAST;

7830     if (ind->dl_dest_addr_offset > sizeof (*ind) &&
7831         ind->dl_dest_addr_offset + ind->dl_dest_addr_length < MBLKL(mb) &&
7832         (bmp = ill->ill_bcast_mp) != NULL) {
7833         dl_unitdata_req_t *dlur;
7834         uint8_t *bphys_addr;

7836         dlur = (dl_unitdata_req_t *)bmp->b_rptr;
7837         bphys_addr = (uchar_t *)dlur + dlur->dl_dest_addr_offset +
7838             extra_offset;

7840         if (bcmp(mhip->mhi_daddr, bphys_addr,
7841             ind->dl_dest_addr_length) == 0)
7842             mhip->mhi_dsttype = MAC_ADDRTYPE_BROADCAST;
7843     }
7844 }

7846 /*
7847  * This function is used to construct a mac_header_info_s from a
7848  * M_DATA fastpath message from a DLPI driver.
7849  * The address fields in the mhi structure points into the message,

```

```

7850  * thus the caller can't use those fields after freeing the message.
7851  */
7852  * We determine whether the packet received is a non-unicast packet
7853  * and in doing so, determine whether or not it is broadcast vs multicast.
7854  * For it to be a broadcast packet, we must have the appropriate mblk_t
7855  * hanging off the ill_t. If this is either not present or doesn't match
7856  * the destination mac address in the DL_UNITDATA_IND, the packet is deemed
7857  * to be multicast. Thus NICs that have no broadcast address (or no
7858  * capability for one, such as point to point links) cannot return as
7859  * the packet being broadcast.
7860  */
7861 void
7862 ip_mdata_to_mhi(ill_t *ill, mblk_t *mp, struct mac_header_info_s *mhip)
7863 {
7864     mblk_t *bmp;
7865     struct ether_header *pether;

7867     bzero(mhip, sizeof (struct mac_header_info_s));

7869     mhip->mhi_dsttype = MAC_ADDRTYPE_UNICAST;

7871     pether = (struct ether_header *)((char *)mp->b_rptr
7872         - sizeof (struct ether_header));

7874     /*
7875      * Make sure the interface is an ethernet type, since we don't
7876      * know the header format for anything but Ethernet. Also make
7877      * sure we are pointing correctly above db_base.
7878      */
7879     if (ill->ill_type != IFT_ETHER)
7880         return;

7882 retry:
7883     if ((uchar_t *)pether < mp->b_datap->db_base)
7884         return;

7886     /* Is there a VLAN tag? */
7887     if (ill->ill_isv6) {
7888         if (pether->ether_type != htons(ETHERTYPE_IPV6)) {
7889             pether = (struct ether_header *)((char *)pether - 4);
7890             goto retry;
7891         }
7892     } else {
7893         if (pether->ether_type != htons(ETHERTYPE_IP)) {
7894             pether = (struct ether_header *)((char *)pether - 4);
7895             goto retry;
7896         }
7897     }
7898     mhip->mhi_daddr = (uchar_t *)&pether->ether_dhost;
7899     mhip->mhi_saddr = (uchar_t *)&pether->ether_shost;

7901     if (!(mhip->mhi_daddr[0] & 0x01))
7902         return;

7904     /* Multicast or broadcast */
7905     mhip->mhi_dsttype = MAC_ADDRTYPE_MULTICAST;

7907     if ((bmp = ill->ill_bcast_mp) != NULL) {
7908         dl_unitdata_req_t *dlur;
7909         uint8_t *bphys_addr;
7910         uint_t addrlen;

7912         dlur = (dl_unitdata_req_t *)bmp->b_rptr;
7913         addrlen = dlur->dl_dest_addr_length;
7914         if (ill->ill_sap_length < 0) {
7915             bphys_addr = (uchar_t *)dlur +

```

```

7916         dlur->dl_dest_addr_offset;
7917         addrlen += ill->ill_sap_length;
7918     } else {
7919         bphys_addr = (uchar_t *)dlur +
7920         dlur->dl_dest_addr_offset +
7921         ill->ill_sap_length;
7922         addrlen -= ill->ill_sap_length;
7923     }
7924     if (bcmp(mhip->mhi_daddr, bphys_addr, addrlen) == 0)
7925         mhip->mhi_dsttype = MAC_ADDRTYPE_BROADCAST;
7926 }
7927 }

7929 /*
7930  * Handle anything but M_DATA messages
7931  * We see the DL_UNITDATA_IND which are part
7932  * of the data path, and also the other messages from the driver.
7933  */
7934 void
7935 ip_rput_notdata(ill_t *ill, mblk_t *mp)
7936 {
7937     mblk_t         *first_mp;
7938     struct iocblk   *iocp;
7939     struct mac_header_info_s mhi;

7941     switch (DB_TYPE(mp)) {
7942     case M_PROTO:
7943     case M_PCPROTO: {
7944         if (((dl_unitdata_ind_t *)mp->b_rptr)->dl_primitive !=
7945             DL_UNITDATA_IND) {
7946             /* Go handle anything other than data elsewhere. */
7947             ip_rput_dlp(ill, mp);
7948             return;
7949         }

7951         first_mp = mp;
7952         mp = first_mp->b_cont;
7953         first_mp->b_cont = NULL;

7955         if (mp == NULL) {
7956             freeb(first_mp);
7957             return;
7958         }
7959         ip_dlur_to_mhi(ill, first_mp, &mhi);
7960         if (ill->ill_isv6)
7961             ip_input_v6(ill, NULL, mp, &mhi);
7962         else
7963             ip_input(ill, NULL, mp, &mhi);

7965         /* Ditch the DLPI header. */
7966         freeb(first_mp);
7967         return;
7968     }
7969     case M_IOCACK:
7970         iocp = (struct iocblk *)mp->b_rptr;
7971         switch (iocp->ioc_cmd) {
7972         case DL_IOC_HDR_INFO:
7973             ill_fastpath_ack(ill, mp);
7974             return;
7975         default:
7976             putnext(ill->ill_rq, mp);
7977             return;
7978         }
7979         /* FALLTHRU */
7980     case M_ERROR:
7981     case M_HANGUP:

```

```

7982         mutex_enter(&ill->ill_lock);
7983         if (ill->ill_state_flags & ILL_CONDEMNED) {
7984             mutex_exit(&ill->ill_lock);
7985             freemsg(mp);
7986             return;
7987         }
7988         ill_refhold_locked(ill);
7989         mutex_exit(&ill->ill_lock);
7990         qwriter_ip(ill, ill->ill_rq, mp, ip_rput_other, CUR_OP,
7991             B_FALSE);
7992         return;
7993     case M_CTL:
7994         putnext(ill->ill_rq, mp);
7995         return;
7996     case M_IOCNAK:
7997         ipldbg(("got iocnak "));
7998         iocp = (struct iocblk *)mp->b_rptr;
7999         switch (iocp->ioc_cmd) {
8000         case DL_IOC_HDR_INFO:
8001             ip_rput_other(NULL, ill->ill_rq, mp, NULL);
8002             return;
8003         default:
8004             break;
8005         }
8006         /* FALLTHRU */
8007     default:
8008         putnext(ill->ill_rq, mp);
8009         return;
8010     }
8011 }

8013 /* Read side put procedure.  Packets coming from the wire arrive here. */
8014 void
8015 ip_rput(queue_t *q, mblk_t *mp)
8016 {
8017     ill_t *ill;
8018     union DL_primitives *dl;

8020     ill = (ill_t *)q->q_ptr;

8022     if (ill->ill_state_flags & (ILL_CONDEMNED | ILL_LL_SUBNET_PENDING)) {
8023         /*
8024          * If things are opening or closing, only accept high-priority
8025          * DLPI messages.  (On open ill->ill_ipif has not yet been
8026          * created; on close, things hanging off the ill may have been
8027          * freed already.)
8028          */
8029         dl = (union DL_primitives *)mp->b_rptr;
8030         if (DB_TYPE(mp) != M_PCPROTO ||
8031             dl->dl_primitive == DL_UNITDATA_IND) {
8032             inet_freemsg(mp);
8033             return;
8034         }
8035     }
8036     if (DB_TYPE(mp) == M_DATA) {
8037         struct mac_header_info_s mhi;

8039         ip_mdata_to_mhi(ill, mp, &mhi);
8040         ip_input(ill, NULL, mp, &mhi);
8041     } else {
8042         ip_rput_notdata(ill, mp);
8043     }
8044 }

8046 /*
8047  * Move the information to a copy.

```

```

8048 */
8049 mblk_t *
8050 ip_fix_dbref(mblk_t *mp, ip_recv_attr_t *ira)
8051 {
8052     mblk_t      *mpl;
8053     ill_t        *ill = ira->ira_ill;
8054     ip_stack_t    *ipst = ill->ill_ipst;
8055
8056     IP_STAT(ipst, ip_db_ref);
8057
8058     /* Make sure we have ira_l2src before we loose the original mblk */
8059     if (!(ira->ira_flags & IRAF_L2SRC_SET))
8060         ip_setl2src(mp, ira, ira->ira_rill);
8061
8062     mpl = copymsg(mp);
8063     if (mpl == NULL) {
8064         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
8065         ip_drop_input("ipIfStatsInDiscards", mp, ill);
8066         freemsg(mp);
8067         return (NULL);
8068     }
8069     /* preserve the hardware checksum flags and data, if present */
8070     if (DB_CKSUMFLAGS(mp) != 0) {
8071         DB_CKSUMFLAGS(mpl) = DB_CKSUMFLAGS(mp);
8072         DB_CKSUMSTART(mpl) = DB_CKSUMSTART(mp);
8073         DB_CKSUMSTUFF(mpl) = DB_CKSUMSTUFF(mp);
8074         DB_CKSUMEND(mpl) = DB_CKSUMEND(mp);
8075         DB_CKSUM16(mpl) = DB_CKSUM16(mp);
8076     }
8077     freemsg(mp);
8078     return (mpl);
8079 }
8080
8081 static void
8082 ip_dlpi_error(ill_t *ill, t_uscalar_t prim, t_uscalar_t dl_err,
8083               t_uscalar_t err)
8084 {
8085     if (dl_err == DL_SYSERR) {
8086         (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
8087             "%s: %s failed: DL_SYSERR (errno %u)\n",
8088             ill->ill_name, dl_primstr(prim), err);
8089         return;
8090     }
8091
8092     (void) mi_strlog(ill->ill_rq, 1, SL_CONSOLE|SL_ERROR|SL_TRACE,
8093         "%s: %s failed: %s\n", ill->ill_name, dl_primstr(prim),
8094         dl_errstr(dl_err));
8095 }
8096
8097 /*
8098 * ip_rput_dlpi is called by ip_rput to handle all DLPI messages other
8099 * than DL_UNITDATA_IND messages. If we need to process this message
8100 * exclusively, we call qwriter_ip, in which case we also need to call
8101 * ill_refhold before that, since qwriter_ip does an ill_refrele.
8102 */
8103 void
8104 ip_rput_dlpi(ill_t *ill, mblk_t *mp)
8105 {
8106     dl_ok_ack_t      *dloa = (dl_ok_ack_t *)mp->b_rptr;
8107     dl_error_ack_t    *dlea = (dl_error_ack_t *)dloa;
8108     queue_t           *q = ill->ill_rq;
8109     t_uscalar_t       prim = dloa->dl_primitive;
8110     t_uscalar_t       reqprim = DL_PRIM_INVAL;
8111
8112     DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi",
8113         char *, dl_primstr(prim), ill_t *, ill);

```

```

8114     ipldbg(("ip_rput_dlpi"));
8115
8116     /*
8117      * If we received an ACK but didn't send a request for it, then it
8118      * can't be part of any pending operation; discard up-front.
8119      */
8120     switch (prim) {
8121     case DL_ERROR_ACK:
8122         reqprim = dlea->dl_error_primitive;
8123         ip2dbg(("ip_rput_dlpi(%s): DL_ERROR_ACK for %s (0x%x): %s "
8124             "(0x%x), unix %u\n", ill->ill_name, dl_primstr(reqprim),
8125             reqprim, dl_errstr(dlea->dl_errno), dlea->dl_errno,
8126             dlea->dl_unix_errno));
8127         break;
8128     case DL_OK_ACK:
8129         reqprim = dloa->dl_correct_primitive;
8130         break;
8131     case DL_INFO_ACK:
8132         reqprim = DL_INFO_REQ;
8133         break;
8134     case DL_BIND_ACK:
8135         reqprim = DL_BIND_REQ;
8136         break;
8137     case DL_PHYS_ADDR_ACK:
8138         reqprim = DL_PHYS_ADDR_REQ;
8139         break;
8140     case DL_NOTIFY_ACK:
8141         reqprim = DL_NOTIFY_REQ;
8142         break;
8143     case DL_CAPABILITY_ACK:
8144         reqprim = DL_CAPABILITY_REQ;
8145         break;
8146     }
8147
8148     if (prim != DL_NOTIFY_IND) {
8149         if (reqprim == DL_PRIM_INVAL ||
8150             !ill_dlpi_pending(ill, reqprim)) {
8151             /* Not a DLPI message we support or expected */
8152             freemsg(mp);
8153             return;
8154         }
8155         ipldbg(("ip_rput: received %s for %s\n", dl_primstr(prim),
8156             dl_primstr(reqprim)));
8157     }
8158
8159     switch (reqprim) {
8160     case DL_UNBIND_REQ:
8161         /*
8162          * NOTE: we mark the unbind as complete even if we got a
8163          * DL_ERROR_ACK, since there's not much else we can do.
8164          */
8165         mutex_enter(&ill->ill_lock);
8166         ill->ill_state_flags &= ~ILL_DL_UNBIND_IN_PROGRESS;
8167         cv_signal(&ill->ill_cv);
8168         mutex_exit(&ill->ill_lock);
8169         break;
8170
8171     case DL_ENABMULTI_REQ:
8172         if (prim == DL_OK_ACK) {
8173             if (ill->ill_dlpi_multicast_state == IDS_INPROGRESS)
8174                 ill->ill_dlpi_multicast_state = IDS_OK;
8175         }
8176         break;
8177     }
8178
8179     /*

```

```

8180      * The message is one we're waiting for (or DL_NOTIFY_IND), but we
8181      * need to become writer to continue to process it. Because an
8182      * exclusive operation doesn't complete until replies to all queued
8183      * DLPI messages have been received, we know we're in the middle of an
8184      * exclusive operation and pass CUR_OP (except for DL_NOTIFY_IND).
8185      *
8186      * As required by qwriter_ip(), we refhold the ill; it will refrele.
8187      * Since this is on the ill stream we unconditionally bump up the
8188      * refcount without doing ILL_CAN_LOOKUP().
8189      */
8190      ill_refhold(ill);
8191      if (prim == DL_NOTIFY_IND)
8192          qwriter_ip(ill, q, mp, ip_rput_dlpi_writer, NEW_OP, B_FALSE);
8193      else
8194          qwriter_ip(ill, q, mp, ip_rput_dlpi_writer, CUR_OP, B_FALSE);
8195  }

8197 /*
8198  * Handling of DLPI messages that require exclusive access to the ipsq.
8199  */
8200  * Need to do ipsq_pending_mp_get on ioctl completion, which could
8201  * happen here. (along with mi_copy_done)
8202  */
8203  /* ARGSUSED */
8204  static void
8205  ip_rput_dlpi_writer(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
8206  {
8207      dl_ok_ack_t      *dloa = (dl_ok_ack_t *)mp->b_rptr;
8208      dl_error_ack_t    *dlea = (dl_error_ack_t *)dloa;
8209      int               err = 0;
8210      ill_t             *ill = (ill_t *)q->q_ptr;
8211      ipif_t            *ipif = NULL;
8212      mblk_t            *mpl = NULL;
8213      conn_t            *connp = NULL;
8214      t_uscalar_t       paddrreq;
8215      mblk_t            *mp_hw;
8216      boolean_t         success;
8217      boolean_t         ioctl_aborted = B_FALSE;
8218      boolean_t         log = B_TRUE;

8220      DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi_writer",
8221                   char *, dl_primstr(dloa->dl_primitive), ill_t *, ill);

8223      ipldbg(("ip_rput_dlpi_writer .."));
8224      ASSERT(ipsq->ipsq_xop == ill->ill_phyint->phyint_ipsq->ipsq_xop);
8225      ASSERT(IAM_WRITER_ILL(ill));

8227      ipif = ipsq->ipsq_xop->ipx_pending_ipif;
8228      /*
8229      * The current ioctl could have been aborted by the user and a new
8230      * ioctl to bring up another ill could have started. We could still
8231      * get a response from the driver later.
8232      */
8233      if (ipif != NULL && ipif->ipif_ill != ill)
8234          ioctl_aborted = B_TRUE;

8236      switch (dloa->dl_primitive) {
8237      case DL_ERROR_ACK:
8238          ipldbg(("ip_rput_dlpi_writer: got DL_ERROR_ACK for %s\n",
8239                dl_primstr(dlea->dl_error_primitive)));

8241          DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi_writer error",
8242                       char *, dl_primstr(dlea->dl_error_primitive),
8243                       ill_t *, ill);

8245          switch (dlea->dl_error_primitive) {

```

```

8246      case DL_DISABMULTI_REQ:
8247          ill_dlpi_done(ill, dlea->dl_error_primitive);
8248          break;
8249      case DL_PROMISCON_REQ:
8250      case DL_PROMISCOFF_REQ:
8251      case DL_UNBIND_REQ:
8252      case DL_ATTACH_REQ:
8253      case DL_INFO_REQ:
8254          ill_dlpi_done(ill, dlea->dl_error_primitive);
8255          break;
8256      case DL_NOTIFY_REQ:
8257          ill_dlpi_done(ill, DL_NOTIFY_REQ);
8258          log = B_FALSE;
8259          break;
8260      case DL_PHYS_ADDR_REQ:
8261          /*
8262           * For IPv6 only, there are two additional
8263           * phys_addr_req's sent to the driver to get the
8264           * IPv6 token and lla. This allows IP to acquire
8265           * the hardware address format for a given interface
8266           * without having built in knowledge of the hardware
8267           * address. ill_phys_addr_pend keeps track of the last
8268           * DL_PAR sent so we know which response we are
8269           * dealing with. ill_dlpi_done will update
8270           * ill_phys_addr_pend when it sends the next req.
8271           * We don't complete the IOCTL until all three DL_PARs
8272           * have been attempted, so set *_len to 0 and break.
8273           */
8274          paddrreq = ill->ill_phys_addr_pend;
8275          ill_dlpi_done(ill, DL_PHYS_ADDR_REQ);
8276          if (paddrreq == DL_IPV6_TOKEN) {
8277              ill->ill_token_length = 0;
8278              log = B_FALSE;
8279              break;
8280          } else if (paddrreq == DL_IPV6_LINK_LAYER_ADDR) {
8281              ill->ill_nd_lls_len = 0;
8282              log = B_FALSE;
8283              break;
8284          }
8285          /*
8286           * Something went wrong with the DL_PHYS_ADDR_REQ.
8287           * We presumably have an IOCTL hanging out waiting
8288           * for completion. Find it and complete the IOCTL
8289           * with the error noted.
8290           * However, ill_dl_phys was called on an ill queue
8291           * (from SIOCSLIFNAME), thus conn_pending_ill is not
8292           * set. But the ioctl is known to be pending on ill_wq.
8293           */
8294          if (!ill->ill_ifname_pending)
8295              break;
8296          ill->ill_ifname_pending = 0;
8297          if (!ioctl_aborted)
8298              mpl = ipsq_pending_mp_get(ipsq, &connp);
8299          if (mpl != NULL) {
8300              /*
8301               * This operation (SIOCSLIFNAME) must have
8302               * happened on the ill. Assert there is no conn
8303               */
8304              ASSERT(connp == NULL);
8305              q = ill->ill_wq;
8306          }
8307          break;
8308      case DL_BIND_REQ:
8309          ill_dlpi_done(ill, DL_BIND_REQ);
8310          if (ill->ill_ifname_pending)
8311              break;

```



```

8312     mutex_enter(&ill->ill_lock);
8313     ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
8314     mutex_exit(&ill->ill_lock);
8315     /*
8316      * Something went wrong with the bind. We presumably
8317      * have an IOCTL hanging out waiting for completion.
8318      * Find it, take down the interface that was coming
8319      * up, and complete the IOCTL with the error noted.
8320      */
8321     if (!ioctl_aborted)
8322         mpl = ipsq_pending_mp_get(ipsq, &connp);
8323     if (mpl != NULL) {
8324         /*
8325          * This might be a result of a DL_NOTE_REPLUMB
8326          * notification. In that case, connp is NULL.
8327          */
8328         if (connp != NULL)
8329             q = CONNP_TO_WQ(connp);
8331         (void) ipif_down(ipif, NULL, NULL);
8332         /* error is set below the switch */
8333     }
8334     break;
8335 case DL_ENABMULTI_REQ:
8336     ill_dlpi_done(ill, DL_ENABMULTI_REQ);
8338     if (ill->ill_dlpi_multicast_state == IDS_INPROGRESS)
8339         ill->ill_dlpi_multicast_state = IDS_FAILED;
8340     if (ill->ill_dlpi_multicast_state == IDS_FAILED) {
8342         printf("ip: joining multicasts failed (%d)"
8343              " on %s - will use link layer "
8344              "broadcasts for multicast\n",
8345              dlea->dl_errno, ill->ill_name);
8347         /*
8348          * Set up for multi_bcast; We are the
8349          * writer, so ok to access ill->ill_ipif
8350          * without any lock.
8351          */
8352         mutex_enter(&ill->ill_phyint->phyint_lock);
8353         ill->ill_phyint->phyint_flags |=
8354             PHYI_MULTICAST;
8355         mutex_exit(&ill->ill_phyint->phyint_lock);
8357     }
8358     freemsg(mp); /* Don't want to pass this up */
8359     return;
8360 case DL_CAPABILITY_REQ:
8361     ipldbg(("ip_rput_dlpi_writer: got DL_ERROR_ACK for "
8362           "DL_CAPABILITY_REQ\n"));
8363     if (ill->ill_dlpi_capab_state == IDCS_PROBE_SENT)
8364         ill->ill_dlpi_capab_state = IDCS_FAILED;
8365     ill_capability_done(ill);
8366     freemsg(mp);
8367     return;
8368 }
8369 /*
8370 * Note the error for IOCTL completion (mpl is set when
8371 * ready to complete ioctl). If ill_ifname_pending_err is
8372 * set, an error occurred during plumbing (ill_ifname_pending),
8373 * so we want to report that error.
8374 *
8375 * NOTE: there are two additional DL_PHYS_ADDR_REQ's
8376 * (DL_IPV6_TOKEN and DL_IPV6_LINK_LAYER_ADDR) that are
8377 * expected to get errack'd if the driver doesn't support

```

```

8378     * these flags (e.g. ethernet). log will be set to B_FALSE
8379     * if these error conditions are encountered.
8380     */
8381     if (mpl != NULL) {
8382         if (ill->ill_ifname_pending_err != 0) {
8383             err = ill->ill_ifname_pending_err;
8384             ill->ill_ifname_pending_err = 0;
8385         } else {
8386             err = dlea->dl_unix_errno ?
8387                 dlea->dl_unix_errno : ENXIO;
8388         }
8389     /*
8390      * If we're plumbing an interface and an error hasn't already
8391      * been saved, set ill_ifname_pending_err to the error passed
8392      * up. Ignore the error if log is B_FALSE (see comment above).
8393      */
8394     } else if (log && ill->ill_ifname_pending &&
8395               ill->ill_ifname_pending_err == 0) {
8396         ill->ill_ifname_pending_err = dlea->dl_unix_errno ?
8397             dlea->dl_unix_errno : ENXIO;
8398     }
8400     if (log)
8401         ip_dlpi_error(ill, dlea->dl_error_primitive,
8402                      dlea->dl_errno, dlea->dl_unix_errno);
8403     break;
8404 case DL_CAPABILITY_ACK:
8405     ill_capability_ack(ill, mp);
8406     /*
8407      * The message has been handed off to ill_capability_ack
8408      * and must not be freed below
8409      */
8410     mp = NULL;
8411     break;
8413 case DL_INFO_ACK:
8414     /* Call a routine to handle this one. */
8415     ill_dlpi_done(ill, DL_INFO_REQ);
8416     ip_ll_subnet_defaults(ill, mp);
8417     ASSERT(!MUTEX_HELD(&ill->ill_phyint->phyint_ipsq->ipsq_lock));
8418     return;
8419 case DL_BIND_ACK:
8420     /*
8421      * We should have an IOCTL waiting on this unless
8422      * sent by ill_dl_phys, in which case just return
8423      */
8424     ill_dlpi_done(ill, DL_BIND_REQ);
8426     if (ill->ill_ifname_pending) {
8427         DTRACE_PROBE2(ip_rput_dlpi_ifname_pending,
8428                      ill_t *, ill, mblk_t *, mp);
8429         break;
8430     }
8431     mutex_enter(&ill->ill_lock);
8432     ill->ill_dl_up = 1;
8433     ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
8434     mutex_exit(&ill->ill_lock);
8436     if (!ioctl_aborted)
8437         mpl = ipsq_pending_mp_get(ipsq, &connp);
8438     if (mpl == NULL) {
8439         DTRACE_PROBE1(ip_rput_dlpi_no_mblk, ill_t *, ill);
8440         break;
8441     }
8442     /*
8443      * mpl was added by ill_dl_up(). if that is a result of

```

```

8504         if ((err == 0) && (ill->ill_up_ipifs)) {
8505             err = ill_up_ipifs(ill, q, mpl);
8506             if (err == EINPROGRESS) {
8507                 freemsg(mp);
8508                 return;
8509             }

```

```

8570             ill->ill_current_frag =
8571                 (uint_t)notify->dl_data;
8572             ill->ill_mc_mtu = (uint_t)notify->dl_data;
8573             break;
8574         case DL_NOTE_SDU_SIZE2:
8575             ill->ill_current_frag =

```

```

8576         (uint_t)notify->dl_data1;
8577         ill->ill_mc_mtu = (uint_t)notify->dl_data2;
8578         break;
8579     }
8580     if (ill->ill_current_frag > ill->ill_max_frag)
8581         ill->ill_max_frag = ill->ill_current_frag;

8583     if (!(ill->ill_flags & ILLF_FIXEDMTU)) {
8584         ill->ill_mtu = ill->ill_current_frag;

8586         /*
8587          * If ill_user_mtu was set (via
8588          * SIOCSLIPLNKINFO), clamp ill_mtu at it.
8589          */
8590         if (ill->ill_user_mtu != 0 &&
8591             ill->ill_user_mtu < ill->ill_mtu)
8592             ill->ill_mtu = ill->ill_user_mtu;

8594         if (ill->ill_user_mtu != 0 &&
8595             ill->ill_user_mtu < ill->ill_mc_mtu)
8596             ill->ill_mc_mtu = ill->ill_user_mtu;

8598         if (ill->ill_isv6) {
8599             if (ill->ill_mtu < IPV6_MIN_MTU)
8600                 ill->ill_mtu = IPV6_MIN_MTU;
8601             if (ill->ill_mc_mtu < IPV6_MIN_MTU)
8602                 ill->ill_mc_mtu = IPV6_MIN_MTU;
8603         } else {
8604             if (ill->ill_mtu < IP_MIN_MTU)
8605                 ill->ill_mtu = IP_MIN_MTU;
8606             if (ill->ill_mc_mtu < IP_MIN_MTU)
8607                 ill->ill_mc_mtu = IP_MIN_MTU;
8608         }
8609     } else if (ill->ill_mc_mtu > ill->ill_mtu) {
8610         ill->ill_mc_mtu = ill->ill_mtu;
8611     }

8613     mutex_exit(&ill->ill_lock);
8614     /*
8615      * Make sure all dce_generation checks find out
8616      * that ill_mtu/ill_mc_mtu has changed.
8617      */
8618     if (orig_mtu != ill->ill_mtu ||
8619         orig_mc_mtu != ill->ill_mc_mtu) {
8620         dce_increment_all_generations(ill->ill_isv6,
8621             ill->ill_ipst);
8622     }

8624     /*
8625      * Refresh IPMP meta-interface MTU if necessary.
8626      */
8627     if (IS_UNDER_IPMP(ill))
8628         ipmp_illgrp_refresh_mtu(ill->ill_grp);
8629     break;

8631 case DL_NOTE_LINK_UP:
8632 case DL_NOTE_LINK_DOWN: {
8633     /*
8634      * We are writer. ill / phyint / ipsq assocs stable.
8635      * The RUNNING flag reflects the state of the link.
8636      */
8637     phyint_t *phyint = ill->ill_phyint;
8638     uint64_t new_phyint_flags;
8639     boolean_t changed = B_FALSE;
8640     boolean_t went_up;

```

```

8642     went_up = notify->dl_notification == DL_NOTE_LINK_UP;
8643     mutex_enter(&phyint->phyint_lock);

8645     new_phyint_flags = went_up ?
8646         phyint->phyint_flags | PHYI_RUNNING :
8647         phyint->phyint_flags & ~PHYI_RUNNING;

8649     if (IS_IPMP(ill)) {
8650         new_phyint_flags = went_up ?
8651             new_phyint_flags & ~PHYI_FAILED :
8652             new_phyint_flags | PHYI_FAILED;
8653     }

8655     if (new_phyint_flags != phyint->phyint_flags) {
8656         phyint->phyint_flags = new_phyint_flags;
8657         changed = B_TRUE;
8658     }
8659     mutex_exit(&phyint->phyint_lock);
8660     /*
8661      * ill_restart_dad handles the DAD restart and routing
8662      * socket notification logic.
8663      */
8664     if (changed) {
8665         ill_restart_dad(phyint->phyint_illv4, went_up);
8666         ill_restart_dad(phyint->phyint_illv6, went_up);
8667     }
8668     break;

8669 case DL_NOTE_PROMISC_ON_PHYS: {
8670     phyint_t *phyint = ill->ill_phyint;

8673     mutex_enter(&phyint->phyint_lock);
8674     phyint->phyint_flags |= PHYI_PROMISC;
8675     mutex_exit(&phyint->phyint_lock);
8676     break;
8677 }
8678 case DL_NOTE_PROMISC_OFF_PHYS: {
8679     phyint_t *phyint = ill->ill_phyint;

8681     mutex_enter(&phyint->phyint_lock);
8682     phyint->phyint_flags &= ~PHYI_PROMISC;
8683     mutex_exit(&phyint->phyint_lock);
8684     break;
8685 }
8686 case DL_NOTE_CAPAB_RENEG:
8687     /*
8688      * Something changed on the driver side.
8689      * It wants us to renegotiate the capabilities
8690      * on this ill. One possible cause is the aggregation
8691      * interface under us where a port got added or
8692      * went away.
8693      *
8694      * If the capability negotiation is already done
8695      * or is in progress, reset the capabilities and
8696      * mark the ill's ill_capab_reneg to be B_TRUE,
8697      * so that when the ack comes back, we can start
8698      * the renegotiation process.
8699      *
8700      * Note that if ill_capab_reneg is already B_TRUE
8701      * (ill_dlpi_capab_state is IDS_UNKNOWN in this case),
8702      * the capability resetting request has been sent
8703      * and the renegotiation has not been started yet;
8704      * nothing needs to be done in this case.
8705      */
8706     ipsq_current_start(ipsq, ill->ill_ipif, 0);
8707     ill_capability_reset(ill, B_TRUE);

```

```

8708         ipsq_current_finish(ipsq);
8709         break;

8711     case DL_NOTE_ALLOWED_IPS:
8712         ill_set_allowed_ips(ill, mp);
8713         break;
8714     default:
8715         ip0dbg(("ip_rput_dlpi_writer: unknown notification "
8716             "type 0x%x for DL_NOTIFY_IND\n",
8717             notify->dl_notification));
8718         break;
8719     }

8721     /*
8722     * As this is an asynchronous operation, we
8723     * should not call ill_dlpi_done
8724     */
8725     break;
8726 }
8727 case DL_NOTIFY_ACK: {
8728     dl_notify_ack_t *noteack = (dl_notify_ack_t *)mp->b_rptr;

8730     if (noteack->dl_notifications & DL_NOTE_LINK_UP)
8731         ill->ill_note_link = 1;
8732     ill_dlpi_done(ill, DL_NOTIFY_REQ);
8733     break;
8734 }
8735 case DL_PHYS_ADDR_ACK: {
8736     /*
8737     * As part of plumbing the interface via SIOCSLIFNAME,
8738     * ill_dl_phys() will queue a series of DL_PHYS_ADDR_REQs,
8739     * whose answers we receive here. As each answer is received,
8740     * we call ill_dlpi_done() to dispatch the next request as
8741     * we're processing the current one. Once all answers have
8742     * been received, we use ipsq_pending_mp_get() to dequeue the
8743     * outstanding IOCTL and reply to it. (Because ill_dl_phys()
8744     * is invoked from an ill queue, conn_oper_pending_ill is not
8745     * available, but we know the ioctl is pending on ill_wq.)
8746     */
8747     uint_t paddrrlen, paddrroff;
8748     uint8_t *addr;

8750     paddrreq = ill->ill_phys_addr_pend;
8751     paddrrlen = ((dl_phys_addr_ack_t *)mp->b_rptr)->dl_addr_length;
8752     paddrroff = ((dl_phys_addr_ack_t *)mp->b_rptr)->dl_addr_offset;
8753     addr = mp->b_rptr + paddrroff;

8755     ill_dlpi_done(ill, DL_PHYS_ADDR_REQ);
8756     if (paddrreq == DL_IPV6_TOKEN) {
8757         /*
8758         * bcopy to low-order bits of ill_token
8759         *
8760         * XXX Temporary hack - currently, all known tokens
8761         * are 64 bits, so I'll cheat for the moment.
8762         */
8763         bcopy(addr, &ill->ill_token.s6_addr32[2], paddrrlen);
8764         ill->ill_token_length = paddrrlen;
8765         break;
8766     } else if (paddrreq == DL_IPV6_LINK_LAYER_ADDR) {
8767         ASSERT(ill->ill_nd_lls_mp == NULL);
8768         ill_set_ndmp(ill, mp, paddrroff, paddrrlen);
8769         mp = NULL;
8770         break;
8771     } else if (paddrreq == DL_CURR_DEST_ADDR) {
8772         ASSERT(ill->ill_dest_addr_mp == NULL);
8773         ill->ill_dest_addr_mp = mp;

```

```

8774         ill->ill_dest_addr = addr;
8775         mp = NULL;
8776         if (ill->ill_isv6) {
8777             ill_setdesttoken(ill);
8778             ipif_setdestlinklocal(ill->ill_ipif);
8779         }
8780         break;
8781     }

8783     ASSERT(paddrreq == DL_CURR_PHYS_ADDR);
8784     ASSERT(ill->ill_phys_addr_mp == NULL);
8785     if (!ill->ill_ifname_pending)
8786         break;
8787     ill->ill_ifname_pending = 0;
8788     if (!ioctl_aborted)
8789         mp1 = ipsq_pending_mp_get(ipsq, &connp);
8790     if (mp1 != NULL) {
8791         ASSERT(connp == NULL);
8792         q = ill->ill_wq;
8793     }
8794     /*
8795     * If any error acks received during the plumbing sequence,
8796     * ill_ifname_pending_err will be set. Break out and send up
8797     * the error to the pending ioctl.
8798     */
8799     if (ill->ill_ifname_pending_err != 0) {
8800         err = ill->ill_ifname_pending_err;
8801         ill->ill_ifname_pending_err = 0;
8802         break;
8803     }

8805     ill->ill_phys_addr_mp = mp;
8806     ill->ill_phys_addr = (paddrrlen == 0 ? NULL : addr);
8807     mp = NULL;

8809     /*
8810     * If paddrrlen or ill_phys_addr_length is zero, the DLPI
8811     * provider doesn't support physical addresses. We check both
8812     * paddrrlen and ill_phys_addr_length because sppp (PPP) does
8813     * not have physical addresses, but historically adversises a
8814     * physical address length of 0 in its DL_INFO_ACK, but 6 in
8815     * its DL_PHYS_ADDR_ACK.
8816     */
8817     if (paddrrlen == 0 || ill->ill_phys_addr_length == 0) {
8818         ill->ill_phys_addr = NULL;
8819     } else if (paddrrlen != ill->ill_phys_addr_length) {
8820         ip0dbg(("DL_PHYS_ADDR_ACK: got addr len %d, expected %d",
8821             paddrrlen, ill->ill_phys_addr_length));
8822         err = EINVAL;
8823         break;
8824     }

8826     if (ill->ill_nd_lls_mp == NULL) {
8827         if ((mp_hw = copyb(ill->ill_phys_addr_mp)) == NULL) {
8828             err = ENOMEM;
8829             break;
8830         }
8831         ill_set_ndmp(ill, mp_hw, paddrroff, paddrrlen);
8832     }

8834     if (ill->ill_isv6) {
8835         ill_setdefaulttoken(ill);
8836         ipif_setlinklocal(ill->ill_ipif);
8837     }
8838     break;
8839 }

```

```

8840     case DL_OK_ACK:
8841         ip2dbg(("DL_OK_ACK %s (0x%x)\n",
8842             dl_primstr((int)dloa->dl_correct_primitive),
8843             dloa->dl_correct_primitive));
8844         DTRACE_PROBE3(ill_dlpi, char *, "ip_rput_dlpi_writer ok",
8845             char *, dl_primstr(dloa->dl_correct_primitive),
8846             ill_t *, ill);

8848         switch (dloa->dl_correct_primitive) {
8849             case DL_ENABMULTI_REQ:
8850             case DL_DISABMULTI_REQ:
8851                 ill_dlpi_done(ill, dloa->dl_correct_primitive);
8852                 break;
8853             case DL_PROMISCON_REQ:
8854             case DL_PROMISCOFF_REQ:
8855             case DL_UNBIND_REQ:
8856             case DL_ATTACH_REQ:
8857                 ill_dlpi_done(ill, dloa->dl_correct_primitive);
8858                 break;
8859         }
8860         break;
8861     default:
8862         break;
8863 }

8865 freemsg(mp);
8866 if (mpl == NULL)
8867     return;

8869 /*
8870  * The operation must complete without EINPROGRESS since
8871  * ipsq_pending_mp_get() has removed the mblk (mpl). Otherwise,
8872  * the operation will be stuck forever inside the IPSQ.
8873  */
8874 ASSERT(err != EINPROGRESS);

8876 DTRACE_PROBE4(ipif_ioctl, char *, "ip_rput_dlpi_writer finish",
8877     int, ipsq->ipsq_xop->ipx_current_ioctl, ill_t *, ill,
8878     ipif_t *, NULL);

8880 switch (ipsq->ipsq_xop->ipx_current_ioctl) {
8881     case 0:
8882         ipsq_current_finish(ipsq);
8883         break;

8885     case SIOCSLIFNAME:
8886     case IF_UNITSEL: {
8887         ill_t *ill_other = ILL_OTHER(ill);

8889         /*
8890          * If SIOCSLIFNAME or IF_UNITSEL is about to succeed, and the
8891          * ill has a peer which is in an IPMP group, then place ill
8892          * into the same group. One catch: although ifconfig plumbs
8893          * the appropriate IPMP meta-interface prior to plumbing this
8894          * ill, it is possible for multiple ifconfig applications to
8895          * race (or for another application to adjust plumbing), in
8896          * which case the IPMP meta-interface we need will be missing.
8897          * If so, kick the phyint out of the group.
8898          */
8899         if (err == 0 && ill_other != NULL && IS_UNDER_IPMP(ill_other)) {
8900             ipmp_grp_t *grp = ill->ill_phyint->phyint_grp;
8901             ipmp_illgrp_t *illg;

8903             illg = ill->ill_isv6 ? grp->gr_v6 : grp->gr_v4;
8904             if (illg == NULL)
8905                 ipmp_phyint_leave_grp(ill->ill_phyint);

```

```

8906         else
8907             ipmp_ill_join_illgrp(ill, illg);
8908     }

8910     if (ipsq->ipsq_xop->ipx_current_ioctl == IF_UNITSEL)
8911         ip_ioctl_finish(q, mpl, err, NO_COPYOUT, ipsq);
8912     else
8913         ip_ioctl_finish(q, mpl, err, COPYOUT, ipsq);
8914     break;
8915 }
8916 case SIOCLIFADDIF:
8917     ip_ioctl_finish(q, mpl, err, COPYOUT, ipsq);
8918     break;

8920 default:
8921     ip_ioctl_finish(q, mpl, err, NO_COPYOUT, ipsq);
8922     break;
8923 }
8924 }

8926 /*
8927  * ip_rput_other is called by ip_rput to handle messages modifying the global
8928  * state in IP. If 'ipsq' is non-NULL, caller is writer on it.
8929  */
8930 /* ARGSUSED */
8931 void
8932 ip_rput_other(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
8933 {
8934     ill_t *ill = q->q_ptr;
8935     struct iocblk *iocp;

8937     ipldbg(("ip_rput_other "));
8938     if (ipsq != NULL) {
8939         ASSERT(IAM_WRITER_IPSQ(ipsq));
8940         ASSERT(ipsq->ipsq_xop ==
8941             ill->ill_phyint->phyint_ipsq->ipsq_xop);
8942     }

8944     switch (mp->b_datap->db_type) {
8945     case M_ERROR:
8946     case M_HANGUP:
8947         /*
8948          * The device has a problem. We force the ILL down. It can
8949          * be brought up again manually using SIOCSIFFLAGS (via
8950          * ifconfig or equivalent).
8951          */
8952         ASSERT(ipsq != NULL);
8953         if (mp->b_rptr < mp->b_wptra)
8954             ill->ill_error = (int)(mp->b_rptr & 0xFF);
8955         if (ill->ill_error == 0)
8956             ill->ill_error = ENXIO;
8957         if (!ill_down_start(q, mp))
8958             return;
8959         ipif_all_down_tail(ipsq, q, mp, NULL);
8960         break;
8961     case M_IOCNAK: {
8962         iocp = (struct iocblk *)mp->b_rptr;

8964         ASSERT(iocp->ioc_cmd == DL_IOC_HDR_INFO);
8965         /*
8966          * If this was the first attempt, turn off the fastpath
8967          * probing.
8968          */
8969         mutex_enter(&ill->ill_lock);
8970         if (ill->ill_dlpi_fastpath_state == IDS_INPROGRESS) {
8971             ill->ill_dlpi_fastpath_state = IDS_FAILED;

```

```

8972         mutex_exit(&ill->ill_lock);
8973         /*
8974          * don't flush the nce_t entries: we use them
8975          * as an index to the ncec itself.
8976          */
8977         ipldbg(("ip_rput: DLPI fastpath off on interface %s\n",
8978             ill->ill_name));
8979     } else {
8980         mutex_exit(&ill->ill_lock);
8981     }
8982     freemsg(mp);
8983     break;
8984 }
8985 default:
8986     ASSERT(0);
8987     break;
8988 }
8989 }

8991 /*
8992  * Update any source route, record route or timestamp options
8993  * When it fails it has consumed the message and BUMPed the MIB.
8994  */
8995 boolean_t
8996 ip_forward_options(mblk_t *mp, ipha_t *ipha, ill_t *dst_ill,
8997     ip_rcv_attr_t *ira)
8998 {
8999     ipoptp_t    opts;
9000     uchar_t     *opt;
9001     uint8_t     optval;
9002     uint8_t     optlen;
9003     ipaddr_t    dst;
9004     ipaddr_t    ifaddr;
9005     uint32_t    ts;
9006     timestruc_t now;
9007     ip_stack_t  *ipst = ira->ira_ill->ill_ipst;

9009     ip2dbg(("ip_forward_options\n"));
9010     dst = ipha->ipha_dst;
9011     for (optval = ipoptp_first(&opts, ipha);
9012         optval != IPOPT_EOL;
9013         optval = ipoptp_next(&opts)) {
9014         ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
9015         opt = opts.ipoptp_cur;
9016         optlen = opts.ipoptp_len;
9017         ip2dbg(("ip_forward_options: opt %d, len %d\n",
9018             optval, opts.ipoptp_len));
9019         switch (optval) {
9020             uint32_t off;
9021         case IPOPT_SSRR:
9022         case IPOPT_LSRR:
9023             /* Check if administratively disabled */
9024             if (!ipst->ips_ip_forward_src_routed) {
9025                 BUMP_MIB(dst_ill->ill_ip_mib,
9026                     ipIfStatsForwProhibits);
9027                 ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
9028                     mp, dst_ill);
9029                 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
9030                     ira);
9031                 return (B_FALSE);
9032             }
9033             if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9034                 /*
9035                  * Must be partial since ip_input_options
9036                  * checked for strict.
9037                  */

```

```

9038         break;
9039     }
9040     off = opt[IPOPT_OFFSET];
9041     off--;
9042     redo_srr:
9043     if (optlen < IP_ADDR_LEN ||
9044         off > optlen - IP_ADDR_LEN) {
9045         /* End of source route */
9046         ipldbg(("
9047             ip_forward_options: end of SR\n"));
9048         break;
9049     }
9050     /* Pick a reasonable address on the outbound if */
9051     ASSERT(dst_ill != NULL);
9052     if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9053         INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9054         NULL) != 0) {
9055         /* No source! Shouldn't happen */
9056         ifaddr = INADDR_ANY;
9057     }
9058     bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9059     bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9060     ipldbg(("ip_forward_options: next hop 0x%x\n",
9061         ntohl(dst)));

9063     /*
9064      * Check if our address is present more than
9065      * once as consecutive hops in source route.
9066      */
9067     if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
9068         off += IP_ADDR_LEN;
9069         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9070         goto redo_srr;
9071     }
9072     ipha->ipha_dst = dst;
9073     opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9074     break;
9075 case IPOPT_RR:
9076     off = opt[IPOPT_OFFSET];
9077     off--;
9078     if (optlen < IP_ADDR_LEN ||
9079         off > optlen - IP_ADDR_LEN) {
9080         /* No more room - ignore */
9081         ipldbg(("
9082             ip_forward_options: end of RR\n"));
9083         break;
9084     }
9085     /* Pick a reasonable address on the outbound if */
9086     ASSERT(dst_ill != NULL);
9087     if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9088         INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9089         NULL) != 0) {
9090         /* No source! Shouldn't happen */
9091         ifaddr = INADDR_ANY;
9092     }
9093     bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9094     opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9095     break;
9096 case IPOPT_TS:
9097     /* Insert timestamp if there is room */
9098     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9099     case IPOPT_TS_TSONLY:
9100         off = IPOPT_TS_TIMELEN;
9101         break;
9102     case IPOPT_TS_PRESPEC:
9103     case IPOPT_TS_PRESPEC_RFC791:

```

```

9104         /* Verify that the address matched */
9105         off = opt[IPOPT_OFFSET] - 1;
9106         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9107         if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9108             /* Not for us */
9109             break;
9110         }
9111         /* FALLTHRU */
9112     case IPOPT_TS_TSANDADDR:
9113         off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9114         break;
9115     default:
9116         /*
9117          * ip_put_options should have already
9118          * dropped this packet.
9119          */
9120         cmn_err(CE_PANIC, "ip_forward_options: "
9121             "unknown IT - bug in ip_input_options?\n");
9122         return (B_TRUE); /* Keep "lint" happy */
9123     }
9124     if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9125         /* Increase overflow counter */
9126         off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9127         opt[IPOPT_POS_OV_FLG] =
9128             (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9129                 (off << 4));
9130         break;
9131     }
9132     off = opt[IPOPT_OFFSET] - 1;
9133     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9134     case IPOPT_TS_PRESPEC:
9135     case IPOPT_TS_PRESPEC_RFC791:
9136     case IPOPT_TS_TSANDADDR:
9137         /* Pick a reasonable addr on the outbound if */
9138         ASSERT(dst_ill != NULL);
9139         if (ip_select_source_v4(dst_ill, INADDR_ANY,
9140             dst, INADDR_ANY, ALL_ZONES, ipst, &ifaddr,
9141             NULL, NULL) != 0) {
9142             /* No source! Shouldn't happen */
9143             ifaddr = INADDR_ANY;
9144         }
9145         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9146         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9147         /* FALLTHRU */
9148     case IPOPT_TS_TSONLY:
9149         off = opt[IPOPT_OFFSET] - 1;
9150         /* Compute # of milliseconds since midnight */
9151         gethrtime(&now);
9152         ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9153             now.tv_nsec / (NANOSEC / MILLISEC);
9154         bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9155         opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9156         break;
9157     }
9158     break;
9159 }
9160 return (B_TRUE);
9161 }
9162 }

```

9164 /*

9165 * Call ill_frag_timeout to do garbage collection. ill_frag_timeout

9166 * returns 'true' if there are still fragments left on the queue, in

9167 * which case we restart the timer.

9168 */

9169 void

```

9170 ill_frag_timer(void *arg)
9171 {
9172     ill_t *ill = (ill_t *)arg;
9173     boolean_t frag_pending;
9174     ip_stack_t *ipst = ill->ill_ipst;
9175     time_t timeout;
9176
9177     mutex_enter(&ill->ill_lock);
9178     ASSERT(!ill->ill_fragtimer_executing);
9179     if (ill->ill_state_flags & ILL_CONDEMNED) {
9180         ill->ill_frag_timer_id = 0;
9181         mutex_exit(&ill->ill_lock);
9182         return;
9183     }
9184     ill->ill_fragtimer_executing = 1;
9185     mutex_exit(&ill->ill_lock);
9186
9187     timeout = (ill->ill_isv6 ? ipst->ips_ipv6_reassembly_timeout :
9188         ipst->ips_ip_reassembly_timeout);
9189
9190     frag_pending = ill_frag_timeout(ill, timeout);
9191
9192     /*
9193      * Restart the timer, if we have fragments pending or if someone
9194      * wanted us to be scheduled again.
9195      */
9196     mutex_enter(&ill->ill_lock);
9197     ill->ill_fragtimer_executing = 0;
9198     ill->ill_frag_timer_id = 0;
9199     if (frag_pending || ill->ill_fragtimer_needrestart)
9200         ill_frag_timer_start(ill);
9201     mutex_exit(&ill->ill_lock);
9202 }
9203
9204 void
9205 ill_frag_timer_start(ill_t *ill)
9206 {
9207     ip_stack_t *ipst = ill->ill_ipst;
9208     clock_t timeo_ms;
9209
9210     ASSERT(MUTEX_HELD(&ill->ill_lock));
9211
9212     /* If the ill is closing or opening don't proceed */
9213     if (ill->ill_state_flags & ILL_CONDEMNED)
9214         return;
9215
9216     if (ill->ill_fragtimer_executing) {
9217         /*
9218          * ill_frag_timer is currently executing. Just record the
9219          * the fact that we want the timer to be restarted.
9220          * ill_frag_timer will post a timeout before it returns,
9221          * ensuring it will be called again.
9222          */
9223         ill->ill_fragtimer_needrestart = 1;
9224         return;
9225     }
9226
9227     if (ill->ill_frag_timer_id == 0) {
9228         timeo_ms = (ill->ill_isv6 ? ipst->ips_ipv6_reassembly_timeout :
9229             ipst->ips_ip_reassembly_timeout) * SECONDS;
9230
9231         /*
9232          * The timer is neither running nor is the timeout handler
9233          * executing. Post a timeout so that ill_frag_timer will be
9234          * called
9235          */

```

```
new/usr/src/uts/common/inet/ip/ip.c
```

```

3302         break;
3303     case IPOPT_RR:
3304         off = opt[IPOPT_OFFSET];
3305         off--;
3306         if (optlen < IP_ADDR_LEN ||
3307             off > optlen - IP_ADDR_LEN) {
3308             /* No more room - ignore */
3309             ipldbg((
3310                 "ip_input_local_options: end of RR\n"));
3311             break;
3312         }
3313         /* Pick a reasonable address on the outbound if */
3314         if (ip_select_source_v4(ill, INADDR_ANY, ipha->ipha_dst,
3315             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
3316             NULL) != 0) {
3317             /* No source! Shouldn't happen */
3318             ifaddr = INADDR_ANY;
3319         }
3320         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
3321         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
3322         break;
3323     case IPOPT_TS:
3324         /* Insert timestamp if there is room */
3325         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
3326             case IPOPT_TS_TSONLY:
3327                 off = IPOPT_TS_TIMELEN;
3328                 break;
3329             case IPOPT_TS_PRESPEC:
3330             case IPOPT_TS_PRESPEC_RFC791:
3331                 /* Verify that the address matched */
3332                 off = opt[IPOPT_OFFSET] - 1;
3333                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
3334                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
3335                     /* Not for us */
3336                     break;
3337                 }
3338                 /* FALLTHRU */
3339             case IPOPT_TS_TSANDADDR:
3340                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
3341                 break;
3342             default:
3343                 /*
3344                  * ip_put_options should have already
3345                  * dropped this packet.
3346                  */
3347                 cmn_err(CE_PANIC, "ip_input_local_options: "
3348                     "unknown IT - bug in ip_input_options?\n");
3349                 return (B_TRUE); /* Keep "lint" happy */
3350         }
3351         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
3352             /* Increase overflow counter */
3353             off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
3354             opt[IPOPT_POS_OV_FLG] =
3355                 (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
3356                     (off << 4));
3357             break;
3358         }
3359         off = opt[IPOPT_OFFSET] - 1;
3360         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
3361             case IPOPT_TS_PRESPEC:
3362             case IPOPT_TS_PRESPEC_RFC791:
3363             case IPOPT_TS_TSANDADDR:
3364                 /* Pick a reasonable addr on the outbound if */
3365                 if (ip_select_source_v4(ill, INADDR_ANY,
3366                     ipha->ipha_dst, INADDR_ANY, ALL_ZONES, ipst,
3367                     &ifaddr, NULL, NULL) != 0) {

```



```

3368 /* No source! Shouldn't happen */
3369 ifaddr = INADDR_ANY;
3370 }
3371 bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
3372 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
3373 /* FALLTHRU */
3374 case IPOPT_TS_TSONLY:
3375     off = opt[IPOPT_OFFSET] - 1;
3376     /* Compute # of milliseconds since midnight */
3377     gethrstime(&now);
3378     ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
3379         now.tv_nsec / (NANOSEC / MILLISEC);
3380     bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
3381     opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
3382     break;
3383 }
3384 break;
3385 }
3386 }
3387 return (B_TRUE);

3389 bad_src_route:
3390 /* make sure we clear any indication of a hardware checksum */
3391 DB_CKSUMFLAGS(mp) = 0;
3392 ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
3393 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
3394 return (B_FALSE);

3396 }

3398 /*
3399  * Process IP options in an inbound packet. Always returns the nexthop.
3400  * Normally this is the passed in nexthop, but if there is an option
3401  * that effects the nexthop (such as a source route) that will be returned.
3402  * Sets *errorp if there is an error, in which case an ICMP error has been sent
3403  * and mp freed.
3404  */
3405 ipaddr_t
3406 ip_input_options(ipha_t *ipha, ipaddr_t dst, mblk_t *mp,
3407     ip_recv_attr_t *ira, int *errorp)
3408 {
3409     ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
3410     ipoptp_t        *opts;
3411     uchar_t          *opt;
3412     uint8_t          optval;
3413     uint8_t          optlen;
3414     intptr_t         code = 0;
3415     ire_t            *ire;

3417     ip2dbg(("ip_input_options\n"));
3418     *errorp = 0;
3419     for (optval = ipoptp_first(&opts, ipha);
3420         optval != IPOPT_EOL;
3421         optval = ipoptp_next(&opts)) {
3422         opt = opts.ipoptp_cur;
3423         optlen = opts.ipoptp_len;
3424         ip2dbg(("ip_input_options: opt %d, len %d\n",
3425             optval, optlen));
3426         /*
3427          * Note: we need to verify the checksum before we
3428          * modify anything thus this routine only extracts the next
3429          * hop dst from any source route.
3430          */
3431         switch (optval) {
3432             uint32_t off;
3433             case IPOPT_SSRR:

```

```

9434 case IPOPT_LSRR:
9435     if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9436         if (optval == IPOPT_SSRR) {
9437             ipldbg(("ip_input_options: not next"
9438                 " strict source route 0x%x\n",
9439                 ntohl(dst)));
9440             code = (char *)&ipha->ipha_dst -
9441                 (char *)&ipha
9442             goto param_prob; /* RouterReq's */
9443         }
9444         ip2dbg(("ip_input_options: "
9445             "not next source route 0x%x\n",
9446             ntohl(dst)));
9447         break;
9448     }
9449
9450     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9451         ipldbg(("
9452             "ip_input_options: bad option offset\n"));
9453         code = (char *)&opt[IPOPT_OLEN] -
9454             (char *)&ipha
9455         goto param_prob;
9456     }
9457     off = opt[IPOPT_OFFSET];
9458     off--;
9459     redo_srr:
9460     if (optlen < IP_ADDR_LEN ||
9461         off > optlen - IP_ADDR_LEN) {
9462         /* End of source route */
9463         ipldbg(("ip_input_options: end of SR\n"));
9464         break;
9465     }
9466     bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9467     ipldbg(("ip_input_options: next hop 0x%x\n",
9468         ntohl(dst)));
9469
9470     /*
9471     * Check if our address is present more than
9472     * once as consecutive hops in source route.
9473     * XXX verify per-interface ip_forwarding
9474     * for source route?
9475     */
9476     if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
9477         off += IP_ADDR_LEN;
9478         goto redo_srr;
9479     }
9480
9481     if (dst == htonl(INADDR_LOOPBACK)) {
9482         ipldbg(("ip_input_options: loopback addr in "
9483             "source route!\n"));
9484         goto bad_src_route;
9485     }
9486     /*
9487     * For strict: verify that dst is directly
9488     * reachable.
9489     */
9490     if (optval == IPOPT_SSRR) {
9491         ire = ire_fhtable_lookup_v4(dst, 0, 0,
9492             IRE_INTERFACE, NULL, ALL_ZONES,
9493             ira->ira_tsl,
9494             MATCH_IRE_TYPE | MATCH_IRE_SECATTR, 0, ipst,
9495             NULL);
9496         if (ire == NULL) {
9497             ipldbg(("ip_input_options: SSRR not "
9498                 "directly reachable: 0x%x\n",
9499                 ntohl(dst)));

```

```

9500         goto bad_src_route;
9501     }
9502     ire_refrale(ire);
9503 }
9504 /*
9505  * Defer update of the offset and the record route
9506  * until the packet is forwarded.
9507  */
9508 break;
9509 case IPOPT_RR:
9510     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9511         ipldb("ip_input_options: bad option offset\n");
9512         code = (char *)&opt[IPOPT_OLEN] -
9513             (char *)ipha;
9514         goto param_prob;
9515     }
9516     break;
9517 case IPOPT_TS:
9518     /*
9519      * Verify that length >= 5 and that there is either
9520      * room for another timestamp or that the overflow
9521      * counter is not maxed out.
9522      */
9523     code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
9524     if (optlen < IPOPT_MINLEN_IT) {
9525         goto param_prob;
9526     }
9527     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
9528         ipldb("ip_input_options: bad option offset\n");
9529         code = (char *)&opt[IPOPT_OFFSET] -
9530             (char *)ipha;
9531         goto param_prob;
9532     }
9533     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9534     case IPOPT_TS_TSONLY:
9535         off = IPOPT_TS_TIMELEN;
9536         break;
9537     case IPOPT_TS_TSANDADDR:
9538     case IPOPT_TS_PRESPEC:
9539     case IPOPT_TS_PRESPEC_RFC791:
9540         off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9541         break;
9542     default:
9543         code = (char *)&opt[IPOPT_POS_OV_FLG] -
9544             (char *)ipha;
9545         goto param_prob;
9546     }
9547     if (opt[IPOPT_OFFSET] - 1 + off > optlen &&
9548         (opt[IPOPT_POS_OV_FLG] & 0xF0) == 0xF0) {
9549         /*
9550          * No room and the overflow counter is 15
9551          * already.
9552          */
9553         goto param_prob;
9554     }
9555     break;
9556 }
9557 }
9558 }
9559
9561 if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0) {
9562     return (dst);
9563 }
9565 ipldb("ip_input_options: error processing IP options.");

```

```

9566     code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;
9567
9568 param_prob:
9569     /* make sure we clear any indication of a hardware checksum */
9570     DB_CKSUMFLAGS(mp) = 0;
9571     ip_drop_input("ICMP_PARAM_PROBLEM", mp, ira->ira_ill);
9572     icmp_param_problem(mp, (uint8_t)code, ira);
9573     *errorp = -1;
9574     return (dst);
9575
9576 bad_src_route:
9577     /* make sure we clear any indication of a hardware checksum */
9578     DB_CKSUMFLAGS(mp) = 0;
9579     ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ira->ira_ill);
9580     icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
9581     *errorp = -1;
9582     return (dst);
9583 }
9584
9585 /*
9586  * IP & ICMP info in >=14 msg's ...
9587  * - ip fixed part (mib2_ip_t)
9588  * - icmp fixed part (mib2_icmp_t)
9589  * - ipAddrEntryTable (ip 20) all IPv4 ipifs
9590  * - ipRouteEntryTable (ip 21) all IPv4 IRES
9591  * - ipNetToMediaEntryTable (ip 22) all IPv4 Neighbor Cache entries
9592  * - ipRouteAttributeTable (ip 102) labeled routes
9593  * - ip multicast membership (ip_member_t)
9594  * - ip multicast source filtering (ip_grpsrc_t)
9595  * - igmp fixed part (struct igmpstat)
9596  * - multicast routing stats (struct mrtstat)
9597  * - multicast routing vifs (array of struct vifctl)
9598  * - multicast routing routes (array of struct mfctl)
9599  * - ip6 fixed part (mib2_ipv6IfStatsEntry_t)
9600  * - icmp6 fixed part (mib2_ipv6IfIcmpEntry_t)
9601  * - One per ill plus one generic
9602  * - One per ill plus one generic
9603  * - ipv6RouteEntry all IPv6 IRES
9604  * - ipv6RouteAttributeTable (ip6 102) labeled routes
9605  * - ipv6NetToMediaEntry all IPv6 Neighbor Cache entries
9606  * - ipv6AddrEntry all IPv6 ipifs
9607  * - ipv6 multicast membership (ipv6_member_t)
9608  * - ipv6 multicast source filtering (ipv6_grpsrc_t)
9609  *
9610  * NOTE: original mpctl is copied for msg's 2..N, since its ctl part is
9611  * already filled in by the caller.
9612  * If legacy_req is true then MIB structures needs to be truncated to their
9613  * legacy sizes before being returned.
9614  * Return value of 0 indicates that no messages were sent and caller
9615  * should free mpctl.
9616  */
9617 int
9618 ip_snmp_get(queue_t *q, mblk_t *mpctl, int level, boolean_t legacy_req)
9619 {
9620     ip_stack_t *ipst;
9621     sctp_stack_t *sctps;
9622
9623     if (q->q_next != NULL) {
9624         ipst = ILLQ_TO_IPST(q);
9625     } else {
9626         ipst = CONNQ_TO_IPST(q);
9627     }
9628     ASSERT(ipst != NULL);
9629     sctps = ipst->ips_netstack->netstack_sctp;
9630
9631     if (mpctl == NULL || mpctl->b_cont == NULL) {

```

```

9632         return (0);
9633     }

9635     /*
9636     * For the purposes of the (broken) packet shell use
9637     * of the level we make sure MIB2_TCP/MIB2_UDP can be used
9638     * to make TCP and UDP appear first in the list of mib items.
9639     * TBD: We could expand this and use it in netstat so that
9640     * the kernel doesn't have to produce large tables (connections,
9641     * routes, etc) when netstat only wants the statistics or a particular
9642     * table.
9643     */
9644     if (!(level == MIB2_TCP || level == MIB2_UDP)) {
9645         if ((mpctl = icmp_snmp_get(q, mpctl)) == NULL) {
9646             return (1);
9647         }
9648     }

9650     if (level != MIB2_TCP) {
9651         if ((mpctl = udp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9652             return (1);
9653         }
9654     }

9656     if (level != MIB2_UDP) {
9657         if ((mpctl = tcp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9658             return (1);
9659         }
9660     }

9662     if ((mpctl = ip_snmp_get_mib2_ip_traffic_stats(q, mpctl,
9663         ipst, legacy_req)) == NULL) {
9664         return (1);
9665     }

9667     if ((mpctl = ip_snmp_get_mib2_ip6(q, mpctl, ipst,
9668         legacy_req)) == NULL) {
9669         return (1);
9670     }

9672     if ((mpctl = ip_snmp_get_mib2_icmp(q, mpctl, ipst)) == NULL) {
9673         return (1);
9674     }

9676     if ((mpctl = ip_snmp_get_mib2_icmp6(q, mpctl, ipst)) == NULL) {
9677         return (1);
9678     }

9680     if ((mpctl = ip_snmp_get_mib2_igmp(q, mpctl, ipst)) == NULL) {
9681         return (1);
9682     }

9684     if ((mpctl = ip_snmp_get_mib2_multi(q, mpctl, ipst)) == NULL) {
9685         return (1);
9686     }

9688     if ((mpctl = ip_snmp_get_mib2_ip_addr(q, mpctl, ipst,
9689         legacy_req)) == NULL) {
9690         return (1);
9691     }

9693     if ((mpctl = ip_snmp_get_mib2_ip6_addr(q, mpctl, ipst,
9694         legacy_req)) == NULL) {
9695         return (1);
9696     }

```

```

9698     if ((mpctl = ip_snmp_get_mib2_ip_group_mem(q, mpctl, ipst)) == NULL) {
9699         return (1);
9700     }

9702     if ((mpctl = ip_snmp_get_mib2_ip6_group_mem(q, mpctl, ipst)) == NULL) {
9703         return (1);
9704     }

9706     if ((mpctl = ip_snmp_get_mib2_ip_group_src(q, mpctl, ipst)) == NULL) {
9707         return (1);
9708     }

9710     if ((mpctl = ip_snmp_get_mib2_ip6_group_src(q, mpctl, ipst)) == NULL) {
9711         return (1);
9712     }

9714     if ((mpctl = ip_snmp_get_mib2_virt_multi(q, mpctl, ipst)) == NULL) {
9715         return (1);
9716     }

9718     if ((mpctl = ip_snmp_get_mib2_multi_rtable(q, mpctl, ipst)) == NULL) {
9719         return (1);
9720     }

9722     mpctl = ip_snmp_get_mib2_ip_route_media(q, mpctl, level, ipst);
9723     if (mpctl == NULL)
9724         return (1);

9726     mpctl = ip_snmp_get_mib2_ip6_route_media(q, mpctl, level, ipst);
9727     if (mpctl == NULL)
9728         return (1);

9730     if ((mpctl = sctp_snmp_get_mib2(q, mpctl, sctps)) == NULL) {
9731         return (1);
9732     }

9734 #endif /* ! codereview */
9735     if ((mpctl = ip_snmp_get_mib2_ip_dce(q, mpctl, ipst)) == NULL) {
9736         return (1);
9737     }

9739     if ((mpctl = dccp_snmp_get(q, mpctl, legacy_req)) == NULL) {
9740         return (1);
9741     }

9743 #endif /* ! codereview */
9744     freemsg(mpctl);
9745     return (1);
9746 }

9748 /* Get global (legacy) IPv4 statistics */
9749 static mblk_t *
9750 ip_snmp_get_mib2_ip(queue_t *q, mblk_t *mpctl, mib2_ipIfStatsEntry_t *ipmib,
9751     ip_stack_t *ipst, boolean_t legacy_req)
9752 {
9753     mib2_ip_t          old_ip_mib;
9754     struct opthdr      *optp;
9755     mblk_t              *mp2ctl;
9756     mib2_ipAddrEntry_t mae;

9758     /*
9759     * make a copy of the original message
9760     */
9761     mp2ctl = copymsg(mpctl);

9763     /* fixed length IP structure... */

```

```

9764     optp = (struct ophdr *) &mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9765     optp->level = MIB2_IP;
9766     optp->name = 0;
9767     SET_MIB(old_ip_mib.ipForwarding,
9768             (WE_ARE_FORWARDING(ipst) ? 1 : 2));
9769     SET_MIB(old_ip_mib.ipDefaultTTL,
9770             (uint32_t) ipst->ips_ip_def_ttl);
9771     SET_MIB(old_ip_mib.ipReasmTimeout,
9772             ipst->ips_ip_reassembly_timeout);
9773     SET_MIB(old_ip_mib.ipAddrEntrySize,
9774             (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
9775             sizeof (mib2_ipAddrEntry_t));
9776     SET_MIB(old_ip_mib.ipRouteEntrySize,
9777             sizeof (mib2_ipRouteEntry_t));
9778     SET_MIB(old_ip_mib.ipNetToMediaEntrySize,
9779             sizeof (mib2_ipNetToMediaEntry_t));
9780     SET_MIB(old_ip_mib.ipMemberEntrySize, sizeof (ip_member_t));
9781     SET_MIB(old_ip_mib.ipGroupSourceEntrySize, sizeof (ip_grpsrc_t));
9782     SET_MIB(old_ip_mib.ipRouteAttributeSize,
9783             sizeof (mib2_ipAttributeEntry_t));
9784     SET_MIB(old_ip_mib.transportMLPSize, sizeof (mib2_transportMLPEntry_t));
9785     SET_MIB(old_ip_mib.ipDestEntrySize, sizeof (dest_cache_entry_t));

9787 /*
9788  * Grab the statistics from the new IP MIB
9789  */
9790     SET_MIB(old_ip_mib.ipInReceives,
9791             (uint32_t) ipmib->ipIfStatsHCInReceives);
9792     SET_MIB(old_ip_mib.ipInHdrErrors, ipmib->ipIfStatsInHdrErrors);
9793     SET_MIB(old_ip_mib.ipInAddrErrors, ipmib->ipIfStatsInAddrErrors);
9794     SET_MIB(old_ip_mib.ipForwDatagrams,
9795             (uint32_t) ipmib->ipIfStatsHCOutForwDatagrams);
9796     SET_MIB(old_ip_mib.ipInUnknownProtos,
9797             ipmib->ipIfStatsInUnknownProtos);
9798     SET_MIB(old_ip_mib.ipInDiscards, ipmib->ipIfStatsInDiscards);
9799     SET_MIB(old_ip_mib.ipInDelivers,
9800             (uint32_t) ipmib->ipIfStatsHCInDelivers);
9801     SET_MIB(old_ip_mib.ipOutRequests,
9802             (uint32_t) ipmib->ipIfStatsHCOutRequests);
9803     SET_MIB(old_ip_mib.ipOutDiscards, ipmib->ipIfStatsOutDiscards);
9804     SET_MIB(old_ip_mib.ipOutNoRoutes, ipmib->ipIfStatsOutNoRoutes);
9805     SET_MIB(old_ip_mib.ipReasmReqds, ipmib->ipIfStatsReasmReqds);
9806     SET_MIB(old_ip_mib.ipReasmOKs, ipmib->ipIfStatsReasmOKs);
9807     SET_MIB(old_ip_mib.ipReasmFails, ipmib->ipIfStatsReasmFails);
9808     SET_MIB(old_ip_mib.ipFragOKs, ipmib->ipIfStatsOutFragOKs);
9809     SET_MIB(old_ip_mib.ipFragFails, ipmib->ipIfStatsOutFragFails);
9810     SET_MIB(old_ip_mib.ipFragCreates, ipmib->ipIfStatsOutFragCreates);

9812 /* ipRoutingDiscards is not being used */
9813     SET_MIB(old_ip_mib.ipRoutingDiscards, 0);
9814     SET_MIB(old_ip_mib.tcpInErrs, ipmib->tcpIfStatsInErrs);
9815     SET_MIB(old_ip_mib.udpNoPorts, ipmib->udpIfStatsNoPorts);
9816     SET_MIB(old_ip_mib.ipInCksumErrs, ipmib->ipIfStatsInCksumErrs);
9817     SET_MIB(old_ip_mib.ipReasmDuplicates,
9818             ipmib->ipIfStatsReasmDuplicates);
9819     SET_MIB(old_ip_mib.ipReasmPartDups, ipmib->ipIfStatsReasmPartDups);
9820     SET_MIB(old_ip_mib.ipForwProhibits, ipmib->ipIfStatsForwProhibits);
9821     SET_MIB(old_ip_mib.udpInCksumErrs, ipmib->udpIfStatsInCksumErrs);
9822     SET_MIB(old_ip_mib.udpInOverflows, ipmib->udpIfStatsInOverflows);
9823     SET_MIB(old_ip_mib.rawipInOverflows,
9824             ipmib->rawipIfStatsInOverflows);

9826     SET_MIB(old_ip_mib.ipsecInSucceeded, ipmib->ipsecIfStatsInSucceeded);
9827     SET_MIB(old_ip_mib.ipsecInFailed, ipmib->ipsecIfStatsInFailed);
9828     SET_MIB(old_ip_mib.ipInIPv6, ipmib->ipIfStatsInWrongIPVersion);
9829     SET_MIB(old_ip_mib.ipOutIPv6, ipmib->ipIfStatsOutWrongIPVersion);

```

```

9830     SET_MIB(old_ip_mib.ipOutSwitchIPv6,
9831             ipmib->ipIfStatsOutSwitchIPVersion);

9833     if (!snmp_append_data(mpctl->b_cont, (char *) &old_ip_mib,
9834             (int) sizeof (old_ip_mib))) {
9835         ipldbg(("ip_snmp_get_mib2_ip: failed to allocate %u bytes\n",
9836             (uint_t) sizeof (old_ip_mib)));
9837     }

9839     optp->len = (t_uscalar_t) msgdsize(mpctl->b_cont);
9840     ip3dbg(("ip_snmp_get_mib2_ip: level %d, name %d, len %d\n",
9841             (int) optp->level, (int) optp->name, (int) optp->len));
9842     greply(q, mpctl);
9843     return (mp2ctl);
9844 }

9846 /* Per interface IPv4 statistics */
9847 static mblk_t *
9848 ip_snmp_get_mib2_ip_traffic_stats(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
9849     boolean_t legacy_req)
9850 {
9851     struct ophdr      *optp;
9852     mblk_t             *mp2ctl;
9853     ill_t              *ill;
9854     ill_walk_context_t ctx;
9855     mblk_t             *mp_tail = NULL;
9856     mib2_ipIfStatsEntry_t global_ip_mib;
9857     mib2_ipAddrEntry_t  mae;

9859     /*
9860      * Make a copy of the original message
9861      */
9862     mp2ctl = copymsg(mpctl);

9864     optp = (struct ophdr *) &mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9865     optp->level = MIB2_IP;
9866     optp->name = MIB2_IP_TRAFFIC_STATS;
9867     /* Include "unknown interface" ip_mib */
9868     ipst->ips_ip_mib.ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv4;
9869     ipst->ips_ip_mib.ipIfStatsIfIndex =
9870         MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
9871     SET_MIB(ipst->ips_ip_mib.ipIfStatsForwarding,
9872             (ipst->ips_ip_forwarding ? 1 : 2));
9873     SET_MIB(ipst->ips_ip_mib.ipIfStatsDefaultTTL,
9874             (uint32_t) ipst->ips_ip_def_ttl);
9875     SET_MIB(ipst->ips_ip_mib.ipIfStatsEntrySize,
9876             sizeof (mib2_ipIfStatsEntry_t));
9877     SET_MIB(ipst->ips_ip_mib.ipIfStatsAddrEntrySize,
9878             sizeof (mib2_ipAddrEntry_t));
9879     SET_MIB(ipst->ips_ip_mib.ipIfStatsRouteEntrySize,
9880             sizeof (mib2_ipRouteEntry_t));
9881     SET_MIB(ipst->ips_ip_mib.ipIfStatsNetToMediaEntrySize,
9882             sizeof (mib2_ipNetToMediaEntry_t));
9883     SET_MIB(ipst->ips_ip_mib.ipIfStatsMemberEntrySize,
9884             sizeof (ip_member_t));
9885     SET_MIB(ipst->ips_ip_mib.ipIfStatsGroupSourceEntrySize,
9886             sizeof (ip_grpsrc_t));

9888     bcopy(&ipst->ips_ip_mib, &global_ip_mib, sizeof (global_ip_mib));

9890     if (legacy_req) {
9891         SET_MIB(global_ip_mib.ipIfStatsAddrEntrySize,
9892             LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t));
9893     }

9895     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,

```

```

9896     (char *)&global_ip_mib, (int)sizeof (global_ip_mib))) {
9897         ipldbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9898             "failed to allocate %u bytes\n",
9899             (uint_t)sizeof (global_ip_mib)));
9900     }

9902     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
9903     ill = ILL_START_WALK_V4(&ctx, ipst);
9904     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
9905         ill->ill_ip_mib->ipIfStatsIfIndex =
9906             ill->ill_phyint->phyint_ifindex;
9907         SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
9908             (ipst->ips_ip_forwarding ? 1 : 2));
9909         SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultTTL,
9910             (uint32_t)ipst->ips_ip_def_ttl);

9912         ip_mib2_add_ip_stats(&global_ip_mib, ill->ill_ip_mib);
9913         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
9914             (char *)ill->ill_ip_mib,
9915             (int)sizeof (*ill->ill_ip_mib))) {
9916             ipldbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9917                 "failed to allocate %u bytes\n",
9918                 (uint_t)sizeof (*ill->ill_ip_mib)));
9919         }
9920     }
9921     rw_exit(&ipst->ips_ill_g_lock);

9923     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9924     ip3dbg(("ip_snmp_get_mib2_ip_traffic_stats: "
9925         "level %d, name %d, len %d\n",
9926         (int)optp->level, (int)optp->name, (int)optp->len));
9927     greply(q, mpctl);

9929     if (mp2ctl == NULL)
9930         return (NULL);

9932     return (ip_snmp_get_mib2_ip(q, mp2ctl, &global_ip_mib, ipst,
9933         legacy_req));
9934 }

9936 /* Global IPv4 ICMP statistics */
9937 static mblk_t *
9938 ip_snmp_get_mib2_icmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9939 {
9940     struct opthdr      *optp;
9941     mblk_t              *mp2ctl;

9943     /*
9944      * Make a copy of the original message
9945      */
9946     mp2ctl = copymsg(mpctl);

9948     optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9949     optp->level = MIB2_ICMP;
9950     optp->name = 0;
9951     if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_icmp_mib,
9952         (int)sizeof (ipst->ips_icmp_mib))) {
9953         ipldbg(("ip_snmp_get_mib2_icmp: failed to allocate %u bytes\n",
9954             (uint_t)sizeof (ipst->ips_icmp_mib)));
9955     }
9956     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9957     ip3dbg(("ip_snmp_get_mib2_icmp: level %d, name %d, len %d\n",
9958         (int)optp->level, (int)optp->name, (int)optp->len));
9959     greply(q, mpctl);
9960     return (mp2ctl);
9961 }

```

```

9963 /* Global IPv4 IGMP statistics */
9964 static mblk_t *
9965 ip_snmp_get_mib2_igmp(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9966 {
9967     struct opthdr      *optp;
9968     mblk_t              *mp2ctl;

9970     /*
9971      * make a copy of the original message
9972      */
9973     mp2ctl = copymsg(mpctl);

9975     optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
9976     optp->level = EXPER_IGMP;
9977     optp->name = 0;
9978     if (!snmp_append_data(mpctl->b_cont, (char *)&ipst->ips_igmpstat,
9979         (int)sizeof (ipst->ips_igmpstat))) {
9980         ipldbg(("ip_snmp_get_mib2_igmp: failed to allocate %u bytes\n",
9981             (uint_t)sizeof (ipst->ips_igmpstat)));
9982     }
9983     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
9984     ip3dbg(("ip_snmp_get_mib2_igmp: level %d, name %d, len %d\n",
9985         (int)optp->level, (int)optp->name, (int)optp->len));
9986     greply(q, mpctl);
9987     return (mp2ctl);
9988 }

9990 /* Global IPv4 Multicast Routing statistics */
9991 static mblk_t *
9992 ip_snmp_get_mib2_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
9993 {
9994     struct opthdr      *optp;
9995     mblk_t              *mp2ctl;

9997     /*
9998      * make a copy of the original message
9999      */
10000     mp2ctl = copymsg(mpctl);

10002     optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10003     optp->level = EXPER_DVMRP;
10004     optp->name = 0;
10005     if (!ip_mroute_stats(mpctl->b_cont, ipst)) {
10006         ip0dbg(("ip_mroute_stats: failed\n"));
10007     }
10008     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10009     ip3dbg(("ip_snmp_get_mib2_multi: level %d, name %d, len %d\n",
10010         (int)optp->level, (int)optp->name, (int)optp->len));
10011     greply(q, mpctl);
10012     return (mp2ctl);
10013 }

10015 /* IPv4 address information */
10016 static mblk_t *
10017 ip_snmp_get_mib2_ip_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10018     boolean_t legacy_req)
10019 {
10020     struct opthdr      *optp;
10021     mblk_t              *mp2ctl;
10022     mblk_t              *mp_tail = NULL;
10023     ill_t               *ill;
10024     ipif_t              *ipif;
10025     uint_t              bitval;
10026     mib2_ipAddrEntry_t  mae;
10027     size_t              mae_size;

```

```

10028     zoneid_t      zoneid;
10029     ill_walk_context_t  ctx;

10031     /*
10032      * make a copy of the original message
10033      */
10034     mp2ctl = copymsg(mpctl);

10036     mae_size = (legacy_req) ? LEGACY_MIB_SIZE(&mae, mib2_ipAddrEntry_t) :
10037         sizeof (mib2_ipAddrEntry_t);

10039     /* ipAddrEntryTable */

10041     otp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10042     otp->level = MIB2_IP;
10043     otp->name = MIB2_IP_ADDR;
10044     zoneid = Q_TO_CONN(q)->conn_zoneid;

10046     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10047     ill = ILL_START_WALK_V4(&ctx, ipst);
10048     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10049         for (ipif = ill->ill_ipif; ipif != NULL;
10050              ipif = ipif->ipif_next) {
10051             if (ipif->ipif_zoneid != zoneid &&
10052                 ipif->ipif_zoneid != ALL_ZONES)
10053                 continue;
10054             /* Sum of count from dead IRE_LO* and our current */
10055             mae.ipAdEntInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10056             if (ipif->ipif_ire_local != NULL) {
10057                 mae.ipAdEntInfo.ae_ibcnt +=
10058                     ipif->ipif_ire_local->ire_ib_pkt_count;
10059             }
10060             mae.ipAdEntInfo.ae_obcnt = 0;
10061             mae.ipAdEntInfo.ae_focnt = 0;

10063             ipif_get_name(ipif, mae.ipAdEntIfIndex.o_bytes,
10064                 OCTET_LENGTH);
10065             mae.ipAdEntIfIndex.o_length =
10066                 mi_strlen(mae.ipAdEntIfIndex.o_bytes);
10067             mae.ipAdEntAddr = ipif->ipif_lcl_addr;
10068             mae.ipAdEntNetMask = ipif->ipif_net_mask;
10069             mae.ipAdEntInfo.ae_subnet = ipif->ipif_subnet;
10070             mae.ipAdEntInfo.ae_subnet_len =
10071                 ip_mask_to_plen(ipif->ipif_net_mask);
10072             mae.ipAdEntInfo.ae_src_addr = ipif->ipif_lcl_addr;
10073             for (bitval = 1;
10074                  bitval &&
10075                  !(bitval & ipif->ipif_brd_addr);
10076                  bitval <= 1)
10077                 noop;
10078             mae.ipAdEntBcastAddr = bitval;
10079             mae.ipAdEntReasmMaxSize = IP_MAXPACKET;
10080             mae.ipAdEntInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10081             mae.ipAdEntInfo.ae_metric = ipif->ipif_ill->ill_metric;
10082             mae.ipAdEntInfo.ae_broadcast_addr =
10083                 ipif->ipif_brd_addr;
10084             mae.ipAdEntInfo.ae_pp_dst_addr =
10085                 ipif->ipif_pp_dst_addr;
10086             mae.ipAdEntInfo.ae_flags = ipif->ipif_flags |
10087                 ill->ill_flags | ill->ill_phyint->phyint_flags;
10088             mae.ipAdEntRetransmitTime =
10089                 ill->ill_reachable_retrans_time;

10091             if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10092                 (char *)&mae, (int)mae_size)) {
10093                 ipldbg(("ip_snmp_get_mib2_ip_addr: failed to "

```

```

10094         "allocate %u bytes\n", (uint_t)mae_size));
10095     }
10096 }
10097 }
10098     rw_exit(&ipst->ips_ill_g_lock);

10100     otp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10101     ip3dbg(("ip_snmp_get_mib2_ip_addr: level %d, name %d, len %d\n",
10102         (int)otp->level, (int)otp->name, (int)otp->len));
10103     greply(q, mpctl);
10104     return (mp2ctl);
10105 }

10107 /* IPv6 address information */
10108 static mblk_t *
10109 ip_snmp_get_mib2_ip6_addr(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10110     boolean_t legacy_req)
10111 {
10112     struct opthdr      *otp;
10113     mblk_t             *mp2ctl;
10114     mblk_t             *mp_tail = NULL;
10115     ill_t              *ill;
10116     ipif_t             *ipif;
10117     mib2_ipv6AddrEntry_t mae6;
10118     size_t             mae6_size;
10119     zoneid_t           zoneid;
10120     ill_walk_context_t  ctx;

10122     /*
10123      * make a copy of the original message
10124      */
10125     mp2ctl = copymsg(mpctl);

10127     mae6_size = (legacy_req) ?
10128         LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t) :
10129         sizeof (mib2_ipv6AddrEntry_t);

10131     /* ipv6AddrEntryTable */

10133     otp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10134     otp->level = MIB2_IP6;
10135     otp->name = MIB2_IP6_ADDR;
10136     zoneid = Q_TO_CONN(q)->conn_zoneid;

10138     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10139     ill = ILL_START_WALK_V6(&ctx, ipst);
10140     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10141         for (ipif = ill->ill_ipif; ipif != NULL;
10142              ipif = ipif->ipif_next) {
10143             if (ipif->ipif_zoneid != zoneid &&
10144                 ipif->ipif_zoneid != ALL_ZONES)
10145                 continue;
10146             /* Sum of count from dead IRE_LO* and our current */
10147             mae6.ipv6AddrInfo.ae_ibcnt = ipif->ipif_ib_pkt_count;
10148             if (ipif->ipif_ire_local != NULL) {
10149                 mae6.ipv6AddrInfo.ae_ibcnt +=
10150                     ipif->ipif_ire_local->ire_ib_pkt_count;
10151             }
10152             mae6.ipv6AddrInfo.ae_obcnt = 0;
10153             mae6.ipv6AddrInfo.ae_focnt = 0;

10155             ipif_get_name(ipif, mae6.ipv6AddrIfIndex.o_bytes,
10156                 OCTET_LENGTH);
10157             mae6.ipv6AddrIfIndex.o_length =
10158                 mi_strlen(mae6.ipv6AddrIfIndex.o_bytes);
10159             mae6.ipv6AddrAddress = ipif->ipif_v6lcl_addr;

```

```

10160     mae6.ipv6AddrPfxLength =
10161         ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
10162     mae6.ipv6AddrInfo.ae_subnet = ipif->ipif_v6subnet;
10163     mae6.ipv6AddrInfo.ae_subnet_len =
10164         mae6.ipv6AddrPfxLength;
10165     mae6.ipv6AddrInfo.ae_src_addr = ipif->ipif_v6lcl_addr;

10167     /* Type: stateless(1), stateful(2), unknown(3) */
10168     if (ipif->ipif_flags & IPIF_ADDRCONF)
10169         mae6.ipv6AddrType = 1;
10170     else
10171         mae6.ipv6AddrType = 2;
10172     /* Anycast: true(1), false(2) */
10173     if (ipif->ipif_flags & IPIF_ANYCAST)
10174         mae6.ipv6AddrAnycastFlag = 1;
10175     else
10176         mae6.ipv6AddrAnycastFlag = 2;

10178     /*
10179      * Address status: preferred(1), deprecated(2),
10180      * invalid(3), inaccessible(4), unknown(5)
10181      */
10182     if (ipif->ipif_flags & IPIF_NOLOCAL)
10183         mae6.ipv6AddrStatus = 3;
10184     else if (ipif->ipif_flags & IPIF_DEPRECATED)
10185         mae6.ipv6AddrStatus = 2;
10186     else
10187         mae6.ipv6AddrStatus = 1;
10188     mae6.ipv6AddrInfo.ae_mtu = ipif->ipif_ill->ill_mtu;
10189     mae6.ipv6AddrInfo.ae_metric =
10190         ipif->ipif_ill->ill_metric;
10191     mae6.ipv6AddrInfo.ae_pp_dst_addr =
10192         ipif->ipif_v6pp_dst_addr;
10193     mae6.ipv6AddrInfo.ae_flags = ipif->ipif_flags |
10194         ill->ill_flags | ill->ill_phyint->phyint_flags;
10195     mae6.ipv6AddrReasmMaxSize = IP_MAXPACKET;
10196     mae6.ipv6AddrIdentifier = ill->ill_token;
10197     mae6.ipv6AddrIdentifierLen = ill->ill_token_length;
10198     mae6.ipv6AddrReachableTime = ill->ill_reachable_time;
10199     mae6.ipv6AddrRetransmitTime =
10200         ill->ill_reachable_retrans_time;
10201     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10202         (char *)&mae6, (int)mae6_size)) {
10203         ipldbg(("ip_snmp_get_mib2_ip6_addr: failed to "
10204             "allocate %u bytes\n",
10205             (uint_t)mae6_size));
10206     }
10207 }
10208 }
10209 rw_exit(&ipst->ips_ill_g_lock);

10211     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10212     ip3dbg(("ip_snmp_get_mib2_ip6_addr: level %d, name %d, len %d\n",
10213         (int)optp->level, (int)optp->name, (int)optp->len));
10214     greply(q, mpctl);
10215     return (mp2ctl);
10216 }

10218 /* IPv4 multicast group membership. */
10219 static mblk_t *
10220 ip_snmp_get_mib2_ip_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10221 {
10222     struct ophdr      *optp;
10223     mblk_t             *mp2ctl;
10224     ill_t              *ill;
10225     ipif_t             *ipif;

```

```

10226     ilm_t              *ilm;
10227     ip_member_t        ipm;
10228     mblk_t             *mp_tail = NULL;
10229     ill_walk_context_t  ctx;
10230     zoneid_t           zoneid;

10232     /*
10233      * make a copy of the original message
10234      */
10235     mp2ctl = copymsg(mpctl);
10236     zoneid = Q_TO_CONN(q)->conn_zoneid;

10238     /* ipGroupMember table */
10239     optp = (struct ophdr *)&mpctl->b_rptr[
10240         sizeof (struct T_optmgmt_ack)];
10241     optp->level = MIB2_IP;
10242     optp->name = EXPER_IP_GROUP_MEMBERSHIP;

10244     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10245     ill = ILL_START_WALK_V4(&ctx, ipst);
10246     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10247         /* Make sure the ill isn't going away. */
10248         if (!ill_check_and_refhold(ill))
10249             continue;
10250         rw_exit(&ipst->ips_ill_g_lock);
10251         rw_enter(&ill->ill_mcast_lock, RW_READER);
10252         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10253             if (ilm->ilm_zoneid != zoneid &&
10254                 ilm->ilm_zoneid != ALL_ZONES)
10255                 continue;

10257         /* Is there an ipif for ilm_ifaddr? */
10258         for (ipif = ill->ill_ipif; ipif != NULL;
10259             ipif = ipif->ipif_next) {
10260             if (!IPIF_IS_CONDEMNED(ipif) &&
10261                 ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10262                 ilm->ilm_ifaddr != INADDR_ANY)
10263                 break;
10264         }
10265         if (ipif != NULL) {
10266             ipif_get_name(ipif,
10267                 ipm.ipGroupMemberIfIndex.o_bytes,
10268                 OCTET_LENGTH);
10269         } else {
10270             ill_get_name(ill,
10271                 ipm.ipGroupMemberIfIndex.o_bytes,
10272                 OCTET_LENGTH);
10273         }
10274         ipm.ipGroupMemberIfIndex.o_length =
10275             mi_strlen(ipm.ipGroupMemberIfIndex.o_bytes);

10277         ipm.ipGroupMemberAddress = ilm->ilm_addr;
10278         ipm.ipGroupMemberRefCnt = ilm->ilm_refcnt;
10279         ipm.ipGroupMemberFilterMode = ilm->ilm_fmode;
10280         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10281             (char *)&ipm, (int)sizeof (ipm))) {
10282             ipldbg(("ip_snmp_get_mib2_ip_group: "
10283                 "failed to allocate %u bytes\n",
10284                 (uint_t)sizeof (ipm)));
10285         }
10286     }
10287     rw_exit(&ill->ill_mcast_lock);
10288     ill_refrele(ill);
10289     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10290 }
10291 rw_exit(&ipst->ips_ill_g_lock);

```

```

10292     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10293     ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10294         (int)optp->level, (int)optp->name, (int)optp->len));
10295     greply(q, mpctl);
10296     return (mp2ctl);
10297 }

10299 /* IPv6 multicast group membership. */
10300 static mblk_t *
10301 ip_snmp_get_mib2_ip6_group_mem(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10302 {
10303     struct ophdr      *optp;
10304     mblk_t             *mp2ctl;
10305     ill_t              *ill;
10306     ilm_t              *ilm;
10307     ipv6_member_t      ipm6;
10308     mblk_t             *mp_tail = NULL;
10309     ill_walk_context_t ctx;
10310     zoneid_t           zoneid;

10312     /*
10313      * make a copy of the original message
10314      */
10315     mp2ctl = copymsg(mpctl);
10316     zoneid = Q_TO_CONN(q)->conn_zoneid;

10318     /* ip6GroupMember table */
10319     optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10320     optp->level = MIB2_IP6;
10321     optp->name = EXPER_IP6_GROUP_MEMBERSHIP;

10323     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10324     ill = ILL_START_WALK_V6(&ctx, ipst);
10325     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10326         /* Make sure the ill isn't going away. */
10327         if (!ill_check_and_refhold(ill))
10328             continue;
10329         rw_exit(&ipst->ips_ill_g_lock);
10330         /*
10331          * Normally we don't have any members on under IPMP interfaces.
10332          * We report them as a debugging aid.
10333          */
10334         rw_enter(&ill->ill_mcast_lock, RW_READER);
10335         ipm6.ipv6GroupMemberIfIndex = ill->ill_phyint->phyint_ifindex;
10336         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10337             if (ilm->ilm_zoneid != zoneid &&
10338                 ilm->ilm_zoneid != ALL_ZONES)
10339                 continue; /* not this zone */
10340             ipm6.ipv6GroupMemberAddress = ilm->ilm_v6addr;
10341             ipm6.ipv6GroupMemberRefCnt = ilm->ilm_refcnt;
10342             ipm6.ipv6GroupMemberFilterMode = ilm->ilm_fmode;
10343             if (!snmp_append_data2(mpctl->b_cont,
10344                 &mp_tail,
10345                 (char *)&ipm6, (int)sizeof (ipm6))) {
10346                 ip1dbg(("ip_snmp_get_mib2_ip6_group: "
10347                     "failed to allocate %u bytes\n",
10348                     (uint_t)sizeof (ipm6)));
10349             }
10350             rw_exit(&ill->ill_mcast_lock);
10351             ill_refrele(ill);
10352             rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10353         }
10354     }
10355     rw_exit(&ipst->ips_ill_g_lock);

10357     optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);

```

```

10358     ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10359         (int)optp->level, (int)optp->name, (int)optp->len));
10360     greply(q, mpctl);
10361     return (mp2ctl);
10362 }

10364 /* IP multicast filtered sources */
10365 static mblk_t *
10366 ip_snmp_get_mib2_ip_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10367 {
10368     struct ophdr      *optp;
10369     mblk_t             *mp2ctl;
10370     ill_t              *ill;
10371     ipif_t             *ipif;
10372     ilm_t              *ilm;
10373     ip_grpsrc_t        ips;
10374     mblk_t             *mp_tail = NULL;
10375     ill_walk_context_t ctx;
10376     zoneid_t           zoneid;
10377     int                i;
10378     slist_t            *sl;

10380     /*
10381      * make a copy of the original message
10382      */
10383     mp2ctl = copymsg(mpctl);
10384     zoneid = Q_TO_CONN(q)->conn_zoneid;

10386     /* ipGroupSource table */
10387     optp = (struct ophdr *)&mpctl->b_rptr[
10388         sizeof (struct T_optmgmt_ack)];
10389     optp->level = MIB2_IP;
10390     optp->name = EXPER_IP_GROUP_SOURCES;

10392     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10393     ill = ILL_START_WALK_V4(&ctx, ipst);
10394     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10395         /* Make sure the ill isn't going away. */
10396         if (!ill_check_and_refhold(ill))
10397             continue;
10398         rw_exit(&ipst->ips_ill_g_lock);
10399         rw_enter(&ill->ill_mcast_lock, RW_READER);
10400         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10401             sl = ilm->ilm_filter;
10402             if (ilm->ilm_zoneid != zoneid &&
10403                 ilm->ilm_zoneid != ALL_ZONES)
10404                 continue;
10405             if (SLIST_IS_EMPTY(sl))
10406                 continue;

10408             /* Is there an ipif for ilm_ifaddr? */
10409             for (ipif = ill->ill_ipif; ipif != NULL;
10410                 ipif = ipif->ipif_next) {
10411                 if (!IPIF_IS_CONDEMNED(ipif) &&
10412                     ipif->ipif_lcl_addr == ilm->ilm_ifaddr &&
10413                     ilm->ilm_ifaddr != INADDR_ANY)
10414                     break;
10415             }
10416             if (ipif != NULL) {
10417                 ipif_get_name(ipif,
10418                     ips.ipGroupSourceIfIndex.o_bytes,
10419                     OCTET_LENGTH);
10420             } else {
10421                 ill_get_name(ill,
10422                     ips.ipGroupSourceIfIndex.o_bytes,
10423                     OCTET_LENGTH);

```



```

10424     }
10425     ips.ipGroupSourceIfIndex.o_length =
10426         mi_strlen(ips.ipGroupSourceIfIndex.o_bytes);

10428     ips.ipGroupSourceGroup = ilm->ilm_addr;
10429     for (i = 0; i < sl->sl_numsrc; i++) {
10430         if (!IN6_IS_ADDR_V4MAPPED(&sl->sl_addr[i]))
10431             continue;
10432         IN6_V4MAPPED_TO_IPADDR(&sl->sl_addr[i],
10433             ips.ipGroupSourceAddress);
10434         if (snmp_append_data2(mpctl->b_cont, &mp_tail,
10435             (char *)&ips, (int)sizeof(ips)) == 0) {
10436             ipldbg(("ip_snmp_get_mib2_ip_group_src:"
10437                 " failed to allocate %u bytes\n",
10438                 (uint_t)sizeof(ips)));
10439         }
10440     }
10441     rw_exit(&ill->ill_mcast_lock);
10442     ill_refrele(ill);
10443     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10444 }
10445
10446 rw_exit(&ipst->ips_ill_g_lock);
10447 optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10448 ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10449     (int)optp->level, (int)optp->name, (int)optp->len));
10450 qreply(q, mpctl);
10451 return (mp2ctl);
10452 }

10454 /* IPv6 multicast filtered sources. */
10455 static mblk_t *
10456 ip_snmp_get_mib2_ip6_group_src(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10457 {
10458     struct ophdr      *optp;
10459     mblk_t            *mp2ctl;
10460     ill_t             *ill;
10461     ilm_t             *ilm;
10462     ipv6_grpsrc_t     ips6;
10463     mblk_t            *mp_tail = NULL;
10464     ill_walk_context_t ctx;
10465     zoneid_t          zoneid;
10466     int               i;
10467     slist_t           *sl;

10469     /*
10470      * make a copy of the original message
10471      */
10472     mp2ctl = copymsg(mpctl);
10473     zoneid = Q_TO_CONN(q)->conn_zoneid;

10475     /* ip6GroupMember table */
10476     optp = (struct ophdr *)&mpctl->b_rptr[ sizeof (struct T_optmgmt_ack)];
10477     optp->level = MIB2_IP6;
10478     optp->name = EXPER_IP6_GROUP_SOURCES;

10480     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10481     ill = ILL_START_WALK_V6(&ctx, ipst);
10482     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10483         /* Make sure the ill isn't going away. */
10484         if (!ill_check_and_refhold(ill))
10485             continue;
10486         rw_exit(&ipst->ips_ill_g_lock);
10487         /*
10488          * Normally we don't have any members on under IPMP interfaces.
10489          * We report them as a debugging aid.

```

```

10490     */
10491     rw_enter(&ill->ill_mcast_lock, RW_READER);
10492     ips6.ipv6GroupSourceIfIndex = ill->ill_phyint->phyint_ifindex;
10493     for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {
10494         sl = ilm->ilm_filter;
10495         if (ilm->ilm_zoneid != zoneid &&
10496             ilm->ilm_zoneid != ALL_ZONES)
10497             continue;
10498         if (SLIST_IS_EMPTY(sl))
10499             continue;
10500         ips6.ipv6GroupSourceGroup = ilm->ilm_v6addr;
10501         for (i = 0; i < sl->sl_numsrc; i++) {
10502             ips6.ipv6GroupSourceAddress = sl->sl_addr[i];
10503             if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10504                 (char *)&ips6, (int)sizeof(ips6))) {
10505                 ipldbg(("ip_snmp_get_mib2_ip6_"
10506                     "group_src: failed to allocate "
10507                     "%u bytes\n",
10508                     (uint_t)sizeof(ips6)));
10509             }
10510         }
10511         rw_exit(&ill->ill_mcast_lock);
10512         ill_refrele(ill);
10513         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10514     }
10515     rw_exit(&ipst->ips_ill_g_lock);

10518     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10519     ip3dbg(("ip_snmp_get: level %d, name %d, len %d\n",
10520         (int)optp->level, (int)optp->name, (int)optp->len));
10521     qreply(q, mpctl);
10522     return (mp2ctl);
10523 }

10525 /* Multicast routing virtual interface table. */
10526 static mblk_t *
10527 ip_snmp_get_mib2_virt_multi(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10528 {
10529     struct ophdr      *optp;
10530     mblk_t            *mp2ctl;

10532     /*
10533      * make a copy of the original message
10534      */
10535     mp2ctl = copymsg(mpctl);

10537     optp = (struct ophdr *)&mpctl->b_rptr[ sizeof (struct T_optmgmt_ack)];
10538     optp->level = EXPER_DVMRP;
10539     optp->name = EXPER_DVMRP_VIF;
10540     if (!ip_mroute_vif(mpctl->b_cont, ipst)) {
10541         ip0dbg(("ip_mroute_vif: failed\n"));
10542     }
10543     optp->len = (t_uscalar_t)msgdsize(mpctl->b_cont);
10544     ip3dbg(("ip_snmp_get_mib2_virt_multi: level %d, name %d, len %d\n",
10545         (int)optp->level, (int)optp->name, (int)optp->len));
10546     qreply(q, mpctl);
10547     return (mp2ctl);
10548 }

10550 /* Multicast routing table. */
10551 static mblk_t *
10552 ip_snmp_get_mib2_multi_rtable(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10553 {
10554     struct ophdr      *optp;
10555     mblk_t            *mp2ctl;

```

```

10557      /*
10558       * make a copy of the original message
10559       */
10560      mp2ctl = copymsg(mpctl);

10562      optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10563      optp->level = EXPER_DVMRP;
10564      optp->name = EXPER_DVMRP_MRT;
10565      if (!ip_mroute_mrt(mpctl->b_cont, ipst)) {
10566          ip0dbg(("ip_mroute_mrt: failed\n"));
10567      }
10568      optp->len = (t_uscalar_t)msgdsz(mpctl->b_cont);
10569      ip3dbg(("ip_snmp_get_mib2_multi_rtable: level %d, name %d, len %d\n",
10570          (int)optp->level, (int)optp->name, (int)optp->len));
10571      greply(q, mpctl);
10572      return (mp2ctl);
10573 }

10575 /*
10576  * Return ipRouteEntryTable, ipNetToMediaEntryTable, and ipRouteAttributeTable
10577  * in one IRE walk.
10578  */
10579 static mblk_t *
10580 ip_snmp_get_mib2_ip_route_media(queue_t *q, mblk_t *mpctl, int level,
10581     ip_stack_t *ipst)
10582 {
10583     struct ophdr    *optp;
10584     mblk_t          *mp2ctl;      /* Returned */
10585     mblk_t          *mp3ctl;      /* nettomedia */
10586     mblk_t          *mp4ctl;      /* routeattrs */
10587     iproutedata_t   ird;
10588     zoneid_t        zoneid;

10590     /*
10591      * make copies of the original message
10592      * - mp2ctl is returned unchanged to the caller for his use
10593      * - mpctl is sent upstream as ipRouteEntryTable
10594      * - mp3ctl is sent upstream as ipNetToMediaEntryTable
10595      * - mp4ctl is sent upstream as ipRouteAttributeTable
10596      */
10597     mp2ctl = copymsg(mpctl);
10598     mp3ctl = copymsg(mpctl);
10599     mp4ctl = copymsg(mpctl);
10600     if (mp3ctl == NULL || mp4ctl == NULL) {
10601         freemsg(mp4ctl);
10602         freemsg(mp3ctl);
10603         freemsg(mp2ctl);
10604         freemsg(mpctl);
10605         return (NULL);
10606     }

10608     bzero(&ird, sizeof (ird));

10610     ird.ird_route.lp_head = mpctl->b_cont;
10611     ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10612     ird.ird_attrs.lp_head = mp4ctl->b_cont;
10613     /*
10614      * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10615      * then also include ire_testhidden IRES and IRE_IF_CLONE. This is
10616      * intended a temporary solution until a proper MIB API is provided
10617      * that provides complete filtering/caller-opt-in.
10618      */
10619     if (level == EXPER_IP_AND_ALL_IRES)
10620         ird.ird_flags |= IRD_REPORT_ALL;

```

```

10622     zoneid = Q_TO_CONN(q)->conn_zoneid;
10623     ire_walk_v4(ip_snmp_get2_v4, &ird, zoneid, ipst);

10625     /* ipRouteEntryTable in mpctl */
10626     optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10627     optp->level = MIB2_IP;
10628     optp->name = MIB2_IP_ROUTE;
10629     optp->len = msgdsz(ird.ird_route.lp_head);
10630     ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10631         (int)optp->level, (int)optp->name, (int)optp->len));
10632     greply(q, mpctl);

10634     /* ipNetToMediaEntryTable in mp3ctl */
10635     ncec_walk(NULL, ip_snmp_get2_v4_media, &ird, ipst);

10637     optp = (struct ophdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10638     optp->level = MIB2_IP;
10639     optp->name = MIB2_IP_MEDIA;
10640     optp->len = msgdsz(ird.ird_netmedia.lp_head);
10641     ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10642         (int)optp->level, (int)optp->name, (int)optp->len));
10643     greply(q, mp3ctl);

10645     /* ipRouteAttributeTable in mp4ctl */
10646     optp = (struct ophdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10647     optp->level = MIB2_IP;
10648     optp->name = EXPER_IP_RTATTR;
10649     optp->len = msgdsz(ird.ird_attrs.lp_head);
10650     ip3dbg(("ip_snmp_get_mib2_ip_route_media: level %d, name %d, len %d\n",
10651         (int)optp->level, (int)optp->name, (int)optp->len));
10652     if (optp->len == 0)
10653         freemsg(mp4ctl);
10654     else
10655         greply(q, mp4ctl);

10657     return (mp2ctl);
10658 }

10660 /*
10661  * Return ipv6RouteEntryTable and ipv6RouteAttributeTable in one IRE walk, and
10662  * ipv6NetToMediaEntryTable in an NDP walk.
10663  */
10664 static mblk_t *
10665 ip_snmp_get_mib2_ip6_route_media(queue_t *q, mblk_t *mpctl, int level,
10666     ip_stack_t *ipst)
10667 {
10668     struct ophdr    *optp;
10669     mblk_t          *mp2ctl;      /* Returned */
10670     mblk_t          *mp3ctl;      /* nettomedia */
10671     mblk_t          *mp4ctl;      /* routeattrs */
10672     iproutedata_t   ird;
10673     zoneid_t        zoneid;

10675     /*
10676      * make copies of the original message
10677      * - mp2ctl is returned unchanged to the caller for his use
10678      * - mpctl is sent upstream as ipv6RouteEntryTable
10679      * - mp3ctl is sent upstream as ipv6NetToMediaEntryTable
10680      * - mp4ctl is sent upstream as ipv6RouteAttributeTable
10681      */
10682     mp2ctl = copymsg(mpctl);
10683     mp3ctl = copymsg(mpctl);
10684     mp4ctl = copymsg(mpctl);
10685     if (mp3ctl == NULL || mp4ctl == NULL) {
10686         freemsg(mp4ctl);
10687         freemsg(mp3ctl);

```

```

10688         freemsg(mp2ctl);
10689         freemsg(mpctl);
10690         return (NULL);
10691     }

10693     bzero(&ird, sizeof (ird));

10695     ird.ird_route.lp_head = mpctl->b_cont;
10696     ird.ird_netmedia.lp_head = mp3ctl->b_cont;
10697     ird.ird_attrslp_head = mp4ctl->b_cont;
10698     /*
10699      * If the level has been set the special EXPER_IP_AND_ALL_IRES value,
10700      * then also include ire_testhidden IRES and IRE_IF_CLONE. This is
10701      * intended a temporary solution until a proper MIB API is provided
10702      * that provides complete filtering/caller-opt-in.
10703      */
10704     if (level == EXPER_IP_AND_ALL_IRES)
10705         ird.ird_flags |= IRD_REPORT_ALL;

10707     zoneid = Q_TO_CONN(q)->conn_zoneid;
10708     ire_walk_v6(ip_snmp_get2_v6_route, &ird, zoneid, ipst);

10710     otp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10711     otp->level = MIB2_IP6;
10712     otp->name = MIB2_IP6_ROUTE;
10713     otp->len = msgdsize(ird.ird_route.lp_head);
10714     ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10715         (int)otp->level, (int)otp->name, (int)otp->len));
10716     qreply(q, mpctl);

10718     /* ipv6NetToMediaEntryTable in mp3ctl */
10719     ncec_walk(NULL, ip_snmp_get2_v6_media, &ird, ipst);

10721     otp = (struct opthdr *)&mp3ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10722     otp->level = MIB2_IP6;
10723     otp->name = MIB2_IP6_MEDIA;
10724     otp->len = msgdsize(ird.ird_netmedia.lp_head);
10725     ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10726         (int)otp->level, (int)otp->name, (int)otp->len));
10727     qreply(q, mp3ctl);

10729     /* ipv6RouteAttributeTable in mp4ctl */
10730     otp = (struct opthdr *)&mp4ctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10731     otp->level = MIB2_IP6;
10732     otp->name = EXPER_IP_RTATTR;
10733     otp->len = msgdsize(ird.ird_attrslp_head);
10734     ip3dbg(("ip_snmp_get_mib2_ip6_route_media: level %d, name %d, len %d\n",
10735         (int)otp->level, (int)otp->name, (int)otp->len));
10736     if (otp->len == 0)
10737         freemsg(mp4ctl);
10738     else
10739         qreply(q, mp4ctl);

10741     return (mp2ctl);
10742 }

10744 /*
10745  * IPv6 mib: One per ill
10746  */
10747 static mblk_t *
10748 ip_snmp_get_mib2_ip6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst,
10749     boolean_t legacy_req)
10750 {
10751     struct opthdr      *otp;
10752     mblk_t             *mp2ctl;
10753     ill_t              *ill;

```

```

10754     ill_walk_context_t  ctx;
10755     mblk_t              *mp_tail = NULL;
10756     mib2_ipv6AddrEntry_t mae6;
10757     mib2_ipIfStatsEntry_t *ise;
10758     size_t              ise_size, iae_size;

10760     /*
10761      * Make a copy of the original message
10762      */
10763     mp2ctl = copymsg(mpctl);

10765     /* fixed length IPv6 structure ... */

10767     if (legacy_req) {
10768         ise_size = LEGACY_MIB_SIZE(&ipst->ips_ip6_mib,
10769             mib2_ipIfStatsEntry_t);
10770         iae_size = LEGACY_MIB_SIZE(&mae6, mib2_ipv6AddrEntry_t);
10771     } else {
10772         ise_size = sizeof (mib2_ipIfStatsEntry_t);
10773         iae_size = sizeof (mib2_ipv6AddrEntry_t);
10774     }

10776     otp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10777     otp->level = MIB2_IP6;
10778     otp->name = 0;
10779     /* Include "unknown interface" ip6_mib */
10780     ipst->ips_ip6_mib.ipIfStatsIPVersion = MIB2_INETADDRESSTYPE_ipv6;
10781     ipst->ips_ip6_mib.ipIfStatsIfIndex =
10782         MIB2_UNKNOWN_INTERFACE; /* Flag to netstat */
10783     SET_MIB(ipst->ips_ip6_mib.ipIfStatsForwarding,
10784         ipst->ips_ipv6_forwarding ? 1 : 2);
10785     SET_MIB(ipst->ips_ip6_mib.ipIfStatsDefaultHopLimit,
10786         ipst->ips_ipv6_def_hops);
10787     SET_MIB(ipst->ips_ip6_mib.ipIfStatsEntrySize,
10788         sizeof (mib2_ipIfStatsEntry_t));
10789     SET_MIB(ipst->ips_ip6_mib.ipIfStatsAddrEntrySize,
10790         sizeof (mib2_ipv6AddrEntry_t));
10791     SET_MIB(ipst->ips_ip6_mib.ipIfStatsRouteEntrySize,
10792         sizeof (mib2_ipv6RouteEntry_t));
10793     SET_MIB(ipst->ips_ip6_mib.ipIfStatsNetToMediaEntrySize,
10794         sizeof (mib2_ipv6NetToMediaEntry_t));
10795     SET_MIB(ipst->ips_ip6_mib.ipIfStatsMemberEntrySize,
10796         sizeof (ipv6_member_t));
10797     SET_MIB(ipst->ips_ip6_mib.ipIfStatsGroupSourceEntrySize,
10798         sizeof (ipv6_grpsrc_t));

10800     /*
10801      * Synchronize 64- and 32-bit counters
10802      */
10803     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInReceives,
10804         ipIfStatsHCInReceives);
10805     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInDelivers,
10806         ipIfStatsHCInDelivers);
10807     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutRequests,
10808         ipIfStatsHCOutRequests);
10809     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutForwDatagrams,
10810         ipIfStatsHCOutForwDatagrams);
10811     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsOutMcastPkts,
10812         ipIfStatsHCOutMcastPkts);
10813     SYNC32_MIB(&ipst->ips_ip6_mib, ipIfStatsInMcastPkts,
10814         ipIfStatsHCInMcastPkts);

10816     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10817         (char *)&ipst->ips_ip6_mib, (int)ise_size)) {
10818         ipldbg(("ip_snmp_get_mib2_ip6: failed to allocate %u bytes\n",
10819             (uint_t)ise_size));

```

```

10820     } else if (legacy_req) {
10821         /* Adjust the EntrySize fields for legacy requests. */
10822         ise =
10823             (mib2_ipIfStatsEntry_t *) (mp_tail->b_wptr - (int) ise_size);
10824         SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10825         SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10826     }

10828     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10829     ill = ILL_START_WALK_V6(&ctx, ipst);
10830     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10831         ill->ill_ip_mib->ipIfStatsIfIndex =
10832             ill->ill_phyint->phyint_ifindex;
10833         SET_MIB(ill->ill_ip_mib->ipIfStatsForwarding,
10834             ipst->ips_ipv6_forwarding ? 1 : 2);
10835         SET_MIB(ill->ill_ip_mib->ipIfStatsDefaultHopLimit,
10836             ill->ill_max_hops);

10838         /*
10839          * Synchronize 64- and 32-bit counters
10840          */
10841         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInReceives,
10842             ipIfStatsHCInReceives);
10843         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInDelivers,
10844             ipIfStatsHCInDelivers);
10845         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutRequests,
10846             ipIfStatsHCOutRequests);
10847         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutForwDatagrams,
10848             ipIfStatsHCOutForwDatagrams);
10849         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsOutMcastPkts,
10850             ipIfStatsHCOutMcastPkts);
10851         SYNC32_MIB(ill->ill_ip_mib, ipIfStatsInMcastPkts,
10852             ipIfStatsHCInMcastPkts);

10854         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10855             (char *) ill->ill_ip_mib, (int) ise_size)) {
10856             ipldbg(("ip_snmp_get_mib2_ip6: failed to allocate "
10857                 "%u bytes\n", (uint_t) ise_size));
10858         } else if (legacy_req) {
10859             /* Adjust the EntrySize fields for legacy requests. */
10860             ise = (mib2_ipIfStatsEntry_t *) (mp_tail->b_wptr -
10861                 (int) ise_size);
10862             SET_MIB(ise->ipIfStatsEntrySize, ise_size);
10863             SET_MIB(ise->ipIfStatsAddrEntrySize, iae_size);
10864         }
10865     }
10866     rw_exit(&ipst->ips_ill_g_lock);

10868     optp->len = (t_uscalar_t) msgdsize(mpctl->b_cont);
10869     ip3dbg(("ip_snmp_get_mib2_ip6: level %d, name %d, len %d\n",
10870         (int) optp->level, (int) optp->name, (int) optp->len));
10871     greply(q, mpctl);
10872     return (mp2ctl);
10873 }

10875 /*
10876  * ICMPv6 mib: One per ill
10877  */
10878 static mblk_t *
10879 ip_snmp_get_mib2_icmp6(queue_t *q, mblk_t *mpctl, ip_stack_t *ipst)
10880 {
10881     struct ophdr      *optp;
10882     mblk_t            *mp2ctl;
10883     ill_t             *ill;
10884     ill_walk_context_t ctx;
10885     mblk_t            *mp_tail = NULL;

```

```

10886     /*
10887      * Make a copy of the original message
10888      */
10889     mp2ctl = copymsg(mpctl);

10891     /* fixed length ICMPv6 structure ... */

10893     optp = (struct ophdr *) &mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
10894     optp->level = MIB2_ICMP6;
10895     optp->name = 0;
10896     /* Include "unknown interface" icmp6_mib */
10897     ipst->ips_icmp6_mib.ipv6IfIcmpIfIndex =
10898         MIB2_UNKNOWN_INTERFACE; /* netstat flag */
10899     ipst->ips_icmp6_mib.ipv6IfIcmpEntrySize =
10900         sizeof (mib2_ipv6IfIcmpEntry_t);
10901     if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10902         (char *) &ipst->ips_icmp6_mib,
10903         (int) sizeof (ipst->ips_icmp6_mib))) {
10904         ipldbg(("ip_snmp_get_mib2_icmp6: failed to allocate %u bytes\n",
10905             (uint_t) sizeof (ipst->ips_icmp6_mib)));
10906     }

10908     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
10909     ill = ILL_START_WALK_V6(&ctx, ipst);
10910     for (; ill != NULL; ill = ill_next(&ctx, ill)) {
10911         ill->ill_icmp6_mib->ipv6IfIcmpIfIndex =
10912             ill->ill_phyint->phyint_ifindex;
10913         if (!snmp_append_data2(mpctl->b_cont, &mp_tail,
10914             (char *) ill->ill_icmp6_mib,
10915             (int) sizeof (*ill->ill_icmp6_mib))) {
10916             ipldbg(("ip_snmp_get_mib2_icmp6: failed to allocate "
10917                 "%u bytes\n",
10918                 (uint_t) sizeof (*ill->ill_icmp6_mib)));
10919         }
10920     }
10921     rw_exit(&ipst->ips_ill_g_lock);

10923     optp->len = (t_uscalar_t) msgdsize(mpctl->b_cont);
10924     ip3dbg(("ip_snmp_get_mib2_icmp6: level %d, name %d, len %d\n",
10925         (int) optp->level, (int) optp->name, (int) optp->len));
10926     greply(q, mpctl);
10927     return (mp2ctl);
10928 }

10930 /*
10931  * ire_walk routine to create both ipRouteEntryTable and
10932  * ipRouteAttributeTable in one IRE walk
10933  */
10934 static void
10935 ip_snmp_get2_v4(ire_t *ire, iproutedata_t *ird)
10936 {
10937     ill_t             *ill;
10938     mib2_ipRouteEntry_t *re;
10939     mib2_ipAttributeEntry_t *aes;
10940     tsol_ire_gw_secattr_t *attrp;
10941     tsol_gc_t         *gc = NULL;
10942     tsol_gcgrp_t       *gcgrp = NULL;
10943     ip_stack_t        *ipst = ire->ire_ipst;

10945     ASSERT(ire->ire_ipversion == IPV4_VERSION);

10947     if (!(ird->ird_flags & IRD_REPORT_ALL)) {
10948         if (ire->ire_testhidden)
10949             return;
10950         if (ire->ire_type & IRE_IF_CLONE)
10951             return;

```

```

10952     }
10953
10954     if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
10955         return;
10956
10957     if ((attrp = ire->ire_gw_secattr) != NULL) {
10958         mutex_enter(&attrp->igsa_lock);
10959         if ((gc = attrp->igsa_gc) != NULL) {
10960             gcgrp = gc->gc_grp;
10961             ASSERT(gcgrp != NULL);
10962             rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
10963         }
10964         mutex_exit(&attrp->igsa_lock);
10965     }
10966     /*
10967     * Return all IRE types for route table... let caller pick and choose
10968     */
10969     re->ipRouteDest = ire->ire_addr;
10970     ill = ire->ire_ill;
10971     re->ipRouteIfIndex.o_length = 0;
10972     if (ill != NULL) {
10973         ill_get_name(ill, re->ipRouteIfIndex.o_bytes, OCTET_LENGTH);
10974         re->ipRouteIfIndex.o_length =
10975             mi_strlen(re->ipRouteIfIndex.o_bytes);
10976     }
10977     re->ipRouteMetric1 = -1;
10978     re->ipRouteMetric2 = -1;
10979     re->ipRouteMetric3 = -1;
10980     re->ipRouteMetric4 = -1;
10981
10982     re->ipRouteNextHop = ire->ire_gateway_addr;
10983     /* indirect(4), direct(3), or invalid(2) */
10984     if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
10985         re->ipRouteType = 2;
10986     else if (ire->ire_type & IRE_ONLINK)
10987         re->ipRouteType = 3;
10988     else
10989         re->ipRouteType = 4;
10990
10991     re->ipRouteProto = -1;
10992     re->ipRouteAge = gethrstime_sec() - ire->ire_create_time;
10993     re->ipRouteMask = ire->ire_mask;
10994     re->ipRouteMetric5 = -1;
10995     re->ipRouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
10996     if (ire->ire_ill != NULL && re->ipRouteInfo.re_max_frag == 0)
10997         re->ipRouteInfo.re_max_frag = ire->ire_ill->ill_mtu;
10998
10999     re->ipRouteInfo.re_frag_flag = 0;
11000     re->ipRouteInfo.re_rtt = 0;
11001     re->ipRouteInfo.re_src_addr = 0;
11002     re->ipRouteInfo.re_ref = ire->ire_refcnt;
11003     re->ipRouteInfo.re_obpkt = ire->ire_ob_pkt_count;
11004     re->ipRouteInfo.re_ibpkt = ire->ire_ib_pkt_count;
11005     re->ipRouteInfo.re_flags = ire->ire_flags;
11006
11007     /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
11008     if (ire->ire_type & IRE_INTERFACE) {
11009         ire_t *child;
11010
11011         rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
11012         child = ire->ire_dep_children;
11013         while (child != NULL) {
11014             re->ipRouteInfo.re_obpkt += child->ire_ob_pkt_count;
11015             re->ipRouteInfo.re_ibpkt += child->ire_ib_pkt_count;
11016             child = child->ire_dep_sib_next;
11017         }

```

```

11018         rw_exit(&ipst->ips_ire_dep_lock);
11019     }
11020
11021     if (ire->ire_flags & RTF_DYNAMIC) {
11022         re->ipRouteInfo.re_ire_type = IRE_HOST_REDIRECT;
11023     } else {
11024         re->ipRouteInfo.re_ire_type = ire->ire_type;
11025     }
11026
11027     if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
11028         (char *)re, (int)sizeof (*re))) {
11029         ipldbg(("ip_snmp_get2_v4: failed to allocate %u bytes\n",
11030             (uint_t)sizeof (*re)));
11031     }
11032
11033     if (gc != NULL) {
11034         iaes.iae_routeidx = ird->ird_idx;
11035         iaes.iae_doi = gc->gc_db->gcdb_doi;
11036         iaes.iae_slrange = gc->gc_db->gcdb_slrange;
11037
11038         if (!snmp_append_data2(ird->ird_attrs.lp_head,
11039             &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
11040             ipldbg(("ip_snmp_get2_v4: failed to allocate %u "
11041                 "bytes\n", (uint_t)sizeof (iaes)));
11042         }
11043     }
11044
11045     /* bump route index for next pass */
11046     ird->ird_idx++;
11047
11048     kmem_free(re, sizeof (*re));
11049     if (gcgrp != NULL)
11050         rw_exit(&gcgrp->gcgrp_rwlock);
11051 }
11052
11053 /*
11054 * ire_walk routine to create ipv6RouteEntryTable and ipRouteEntryTable.
11055 */
11056 static void
11057 ip_snmp_get2_v6_route(ire_t *ire, iproutedata_t *ird)
11058 {
11059     ill_t *ill;
11060     mib2_ipv6RouteEntry_t *re;
11061     mib2_ipAttributeEntry_t iaes;
11062     tsol_ire_gw_secattr_t *attrp;
11063     tsol_gc_t *gc = NULL;
11064     tsol_gcgrp_t *gcgrp = NULL;
11065     ip_stack_t *ipst = ire->ire_ipst;
11066
11067     ASSERT(ire->ire_ipversion == IPV6_VERSION);
11068
11069     if (!(ird->ird_flags & IRD_REPORT_ALL)) {
11070         if (ire->ire_testhidden)
11071             return;
11072         if (ire->ire_type & IRE_IF_CLONE)
11073             return;
11074     }
11075
11076     if ((re = kmem_zalloc(sizeof (*re), KM_NOSLEEP)) == NULL)
11077         return;
11078
11079     if ((attrp = ire->ire_gw_secattr) != NULL) {
11080         mutex_enter(&attrp->igsa_lock);
11081         if ((gc = attrp->igsa_gc) != NULL) {
11082             gcgrp = gc->gc_grp;
11083             ASSERT(gcgrp != NULL);

```

```

11084         rw_enter(&gcgrp->gcgrp_rwlock, RW_READER);
11085     }
11086     mutex_exit(&attrp->igsa_lock);
11087 }
11088 /*
11089  * Return all IRE types for route table... let caller pick and choose
11090  */
11091 re->ipv6RouteDest = ire->ire_addr_v6;
11092 re->ipv6RoutePfxLength = ip_mask_to_plen_v6(&ire->ire_mask_v6);
11093 re->ipv6RouteIndex = 0; /* Unique when multiple with same dest/plen */
11094 re->ipv6RouteIfIndex.o_length = 0;
11095 ill = ire->ire_ill;
11096 if (ill != NULL) {
11097     ill_get_name(ill, re->ipv6RouteIfIndex.o_bytes, OCTET_LENGTH);
11098     re->ipv6RouteIfIndex.o_length =
11099         mi_strlen(re->ipv6RouteIfIndex.o_bytes);
11100 }
11102 ASSERT(!(ire->ire_type & IRE_BROADCAST));
11104 mutex_enter(&ire->ire_lock);
11105 re->ipv6RouteNextHop = ire->ire_gateway_addr_v6;
11106 mutex_exit(&ire->ire_lock);
11108 /* remote(4), local(3), or discard(2) */
11109 if (ire->ire_flags & (RTF_REJECT | RTF_BLACKHOLE))
11110     re->ipv6RouteType = 2;
11111 else if (ire->ire_type & IRE_ONLINK)
11112     re->ipv6RouteType = 3;
11113 else
11114     re->ipv6RouteType = 4;
11116 re->ipv6RouteProtocol = -1;
11117 re->ipv6RoutePolicy = 0;
11118 re->ipv6RouteAge = gethrestime_sec() - ire->ire_create_time;
11119 re->ipv6RouteNextHopRDI = 0;
11120 re->ipv6RouteWeight = 0;
11121 re->ipv6RouteMetric = 0;
11122 re->ipv6RouteInfo.re_max_frag = ire->ire_metrics.iulp_mtu;
11123 if (ire->ire_ill != NULL && re->ipv6RouteInfo.re_max_frag == 0)
11124     re->ipv6RouteInfo.re_max_frag = ire->ire_ill->ill_mtu;
11126 re->ipv6RouteInfo.re_frag_flag = 0;
11127 re->ipv6RouteInfo.re_rtt = 0;
11128 re->ipv6RouteInfo.re_src_addr = ipv6_all_zeros;
11129 re->ipv6RouteInfo.re_obpkt = ire->ire_ob_pkt_count;
11130 re->ipv6RouteInfo.re_ibpkt = ire->ire_ib_pkt_count;
11131 re->ipv6RouteInfo.re_ref = ire->ire_refcnt;
11132 re->ipv6RouteInfo.re_flags = ire->ire_flags;
11134 /* Add the IRE_IF_CLONE's counters to their parent IRE_INTERFACE */
11135 if (ire->ire_type & IRE_INTERFACE) {
11136     ire_t *child;
11138     rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
11139     child = ire->ire_dep_children;
11140     while (child != NULL) {
11141         re->ipv6RouteInfo.re_obpkt += child->ire_ob_pkt_count;
11142         re->ipv6RouteInfo.re_ibpkt += child->ire_ib_pkt_count;
11143         child = child->ire_dep_sib_next;
11144     }
11145     rw_exit(&ipst->ips_ire_dep_lock);
11146 }
11147 if (ire->ire_flags & RTF_DYNAMIC) {
11148     re->ipv6RouteInfo.re_ire_type = IRE_HOST_REDIRECT;
11149 } else {

```

```

11150         re->ipv6RouteInfo.re_ire_type = ire->ire_type;
11151     }
11153     if (!snmp_append_data2(ird->ird_route.lp_head, &ird->ird_route.lp_tail,
11154         (char *)re, (int)sizeof (*re))) {
11155         ipldbg(("ip_snmp_get2_v6: failed to allocate %u bytes\n",
11156             (uint_t)sizeof (*re)));
11157     }
11159     if (gc != NULL) {
11160         iaes.iae_routeidx = ird->ird_idx;
11161         iaes.iae_doi = gc->gc_db->gcdb_doi;
11162         iaes.iae_slrange = gc->gc_db->gcdb_slrange;
11164         if (!snmp_append_data2(ird->ird_attrs.lp_head,
11165             &ird->ird_attrs.lp_tail, (char *)&iaes, sizeof (iaes))) {
11166             ipldbg(("ip_snmp_get2_v6: failed to allocate %u "
11167                 "bytes\n", (uint_t)sizeof (iaes)));
11168         }
11169     }
11171     /* bump route index for next pass */
11172     ird->ird_idx++;
11174     kmem_free(re, sizeof (*re));
11175     if (gcgrp != NULL)
11176         rw_exit(&gcgrp->gcgrp_rwlock);
11177 }
11179 /*
11180  * ncec_walk routine to create ipv6NetToMediaEntryTable
11181  */
11182 static int
11183 ip_snmp_get2_v6_media(ncec_t *nccec, iproutedata_t *ird)
11184 {
11185     ill_t *ill;
11186     mib2_ipv6NetToMediaEntry_t ntme;
11188     ill = ncec->nccec_ill;
11189     /* skip arpc entries, and loopback ncec entries */
11190     if (ill->ill_isv6 == B_FALSE || ill->ill_net_type == IRE_LOOPBACK)
11191         return (0);
11192     /*
11193      * Neighbor cache entry attached to IRE with on-link
11194      * destination.
11195      * We report all IPMP groups on ncec_ill which is normally the upper.
11196      */
11197     ntme.ipv6NetToMediaIfIndex = ill->ill_phyint->phyint_ifindex;
11198     ntme.ipv6NetToMediaNetAddress = ncec->nccec_addr;
11199     ntme.ipv6NetToMediaPhysAddress.o_length = ill->ill_phys_addr_length;
11200     if (nccec->nccec_lladdr != NULL) {
11201         bcopy(nccec->nccec_lladdr, ntme.ipv6NetToMediaPhysAddress.o_bytes,
11202             ntme.ipv6NetToMediaPhysAddress.o_length);
11203     }
11204     /*
11205      * Note: Returns ND_* states. Should be:
11206      * reachable(1), stale(2), delay(3), probe(4),
11207      * invalid(5), unknown(6)
11208      */
11209     ntme.ipv6NetToMediaState = ncec->nccec_state;
11210     ntme.ipv6NetToMediaLastUpdated = 0;
11212     /* other(1), dynamic(2), static(3), local(4) */
11213     if (NCE_MYADDR(nccec)) {
11214         ntme.ipv6NetToMediaType = 4;
11215     } else if (nccec->nccec_flags & NCE_F_PUBLISH) {

```

```

11216         ntme.ipv6NetToMediaType = 1; /* proxy */
11217     } else if (ncec->ncec_flags & NCE_F_STATIC) {
11218         ntme.ipv6NetToMediaType = 3;
11219     } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST)) {
11220         ntme.ipv6NetToMediaType = 1;
11221     } else {
11222         ntme.ipv6NetToMediaType = 2;
11223     }
11225     if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11226         &ird->ird_netmedia.lp_tail, (char *)&ntme, sizeof (ntme))) {
11227         ipldbg(("ip_snmp_get2_v6_media: failed to allocate %u bytes\n",
11228             (uint_t)sizeof (ntme)));
11229     }
11230     return (0);
11231 }

11233 int
11234 nce2ace(ncec_t *ncec)
11235 {
11236     int flags = 0;

11238     if (NCE_ISREACHABLE(ncec))
11239         flags |= ACE_F_RESOLVED;
11240     if (ncec->ncec_flags & NCE_F_AUTHORITY)
11241         flags |= ACE_F_AUTHORITY;
11242     if (ncec->ncec_flags & NCE_F_PUBLISH)
11243         flags |= ACE_F_PUBLISH;
11244     if ((ncec->ncec_flags & NCE_F_NONUD) != 0)
11245         flags |= ACE_F_PERMANENT;
11246     if (NCE_MYADDR(ncec))
11247         flags |= (ACE_F_MYADDR | ACE_F_AUTHORITY);
11248     if (ncec->ncec_flags & NCE_F_UNVERIFIED)
11249         flags |= ACE_F_UNVERIFIED;
11250     if (ncec->ncec_flags & NCE_F_AUTHORITY)
11251         flags |= ACE_F_AUTHORITY;
11252     if (ncec->ncec_flags & NCE_F_DELAYED)
11253         flags |= ACE_F_DELAYED;
11254     return (flags);
11255 }

11257 /*
11258  * ncec_walk routine to create ipNetToMediaEntryTable
11259  */
11260 static int
11261 ip_snmp_get2_v4_media(ncec_t *ncec, iproutedata_t *ird)
11262 {
11263     ill_t                *ill;
11264     mib2_ipNetToMediaEntry_t ntme;
11265     const char            *name = "unknown";
11266     ipaddr_t              ncec_addr;

11268     ill = ncec->ncec_ill;
11269     if (ill->ill_isv6 || (ncec->ncec_flags & NCE_F_BCAST) ||
11270         ill->ill_net_type == IRE_LOOPBACK)
11271         return (0);

11273     /* We report all IPMP groups on ncec_ill which is normally the upper. */
11274     name = ill->ill_name;
11275     /* Based on RFC 4293: other(1), inval(2), dyn(3), stat(4) */
11276     if (NCE_MYADDR(ncec)) {
11277         ntme.ipNetToMediaType = 4;
11278     } else if (ncec->ncec_flags & (NCE_F_MCAST|NCE_F_BCAST|NCE_F_PUBLISH)) {
11279         ntme.ipNetToMediaType = 1;
11280     } else {
11281         ntme.ipNetToMediaType = 3;

```

```

11282     }
11283     ntme.ipNetToMediaIfIndex.o_length = MIN(OCTET_LENGTH, strlen(name));
11284     bcopy(name, ntme.ipNetToMediaIfIndex.o_bytes,
11285         ntme.ipNetToMediaIfIndex.o_length);

11287     IN6_V4MAPPED_TO_IPADDR(&ncec->ncec_addr, ncec_addr);
11288     bcopy(&ncec_addr, &ntme.ipNetToMediaNetAddress, sizeof (ncec_addr));

11290     ntme.ipNetToMediaInfo.ntm_mask.o_length = sizeof (ipaddr_t);
11291     ncec_addr = INADDR_BROADCAST;
11292     bcopy(&ncec_addr, ntme.ipNetToMediaInfo.ntm_mask.o_bytes,
11293         sizeof (ncec_addr));
11294     /*
11295      * map all the flags to the ACE counterpart.
11296      */
11297     ntme.ipNetToMediaInfo.ntm_flags = nce2ace(ncec);

11299     ntme.ipNetToMediaPhysAddress.o_length =
11300         MIN(OCTET_LENGTH, ill->ill_phys_addr_length);

11302     if (!NCE_ISREACHABLE(ncec))
11303         ntme.ipNetToMediaPhysAddress.o_length = 0;
11304     else {
11305         if (ncec->ncec_lladdr != NULL) {
11306             bcopy(ncec->ncec_lladdr,
11307                 ntme.ipNetToMediaPhysAddress.o_bytes,
11308                 ntme.ipNetToMediaPhysAddress.o_length);
11309         }
11310     }

11312     if (!snmp_append_data2(ird->ird_netmedia.lp_head,
11313         &ird->ird_netmedia.lp_tail, (char *)&ntme, sizeof (ntme))) {
11314         ipldbg(("ip_snmp_get2_v4_media: failed to allocate %u bytes\n",
11315             (uint_t)sizeof (ntme)));
11316     }
11317     return (0);
11318 }

11320 /*
11321  * return (0) if invalid set request, 1 otherwise, including non-tcp requests
11322  */
11323 /* ARGSUSED */
11324 int
11325 ip_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
11326 {
11327     switch (level) {
11328     case MIB2_IP:
11329     case MIB2_ICMP:
11330         switch (name) {
11331         default:
11332             break;
11333         }
11334         return (1);
11335     default:
11336         return (1);
11337     }
11338 }

11340 /*
11341  * When there exists both a 64- and 32-bit counter of a particular type
11342  * (i.e., InReceives), only the 64-bit counters are added.
11343  */
11344 void
11345 ip_mib2_add_ip_stats(mib2_ipIfStatsEntry_t *o1, mib2_ipIfStatsEntry_t *o2)
11346 {
11347     UPDATE_MIB(o1, ipIfStatsInHdrErrors, o2->ipIfStatsInHdrErrors);

```

```

11348     UPDATE_MIB(o1, ipIfStatsInTooBigErrors, o2->ipIfStatsInTooBigErrors);
11349     UPDATE_MIB(o1, ipIfStatsInNoRoutes, o2->ipIfStatsInNoRoutes);
11350     UPDATE_MIB(o1, ipIfStatsInAddrErrors, o2->ipIfStatsInAddrErrors);
11351     UPDATE_MIB(o1, ipIfStatsInUnknownProtos, o2->ipIfStatsInUnknownProtos);
11352     UPDATE_MIB(o1, ipIfStatsInTruncatedPkts, o2->ipIfStatsInTruncatedPkts);
11353     UPDATE_MIB(o1, ipIfStatsInDiscards, o2->ipIfStatsInDiscards);
11354     UPDATE_MIB(o1, ipIfStatsOutDiscards, o2->ipIfStatsOutDiscards);
11355     UPDATE_MIB(o1, ipIfStatsOutFragOKs, o2->ipIfStatsOutFragOKs);
11356     UPDATE_MIB(o1, ipIfStatsOutFragFails, o2->ipIfStatsOutFragFails);
11357     UPDATE_MIB(o1, ipIfStatsOutFragCreates, o2->ipIfStatsOutFragCreates);
11358     UPDATE_MIB(o1, ipIfStatsReasmReqds, o2->ipIfStatsReasmReqds);
11359     UPDATE_MIB(o1, ipIfStatsReasmOKs, o2->ipIfStatsReasmOKs);
11360     UPDATE_MIB(o1, ipIfStatsReasmFails, o2->ipIfStatsReasmFails);
11361     UPDATE_MIB(o1, ipIfStatsOutNoRoutes, o2->ipIfStatsOutNoRoutes);
11362     UPDATE_MIB(o1, ipIfStatsReasmDuplicates, o2->ipIfStatsReasmDuplicates);
11363     UPDATE_MIB(o1, ipIfStatsReasmPartDups, o2->ipIfStatsReasmPartDups);
11364     UPDATE_MIB(o1, ipIfStatsForwProhibits, o2->ipIfStatsForwProhibits);
11365     UPDATE_MIB(o1, udpInCksumErrs, o2->udpInCksumErrs);
11366     UPDATE_MIB(o1, udpInOverflows, o2->udpInOverflows);
11367     UPDATE_MIB(o1, rawipInOverflows, o2->rawipInOverflows);
11368     UPDATE_MIB(o1, ipIfStatsInWrongIPVersion,
11369         o2->ipIfStatsInWrongIPVersion);
11370     UPDATE_MIB(o1, ipIfStatsOutWrongIPVersion,
11371         o2->ipIfStatsOutWrongIPVersion);
11372     UPDATE_MIB(o1, ipIfStatsOutSwitchIPVersion,
11373         o2->ipIfStatsOutSwitchIPVersion);
11374     UPDATE_MIB(o1, ipIfStatsHCInReceives, o2->ipIfStatsHCInReceives);
11375     UPDATE_MIB(o1, ipIfStatsHCInOctets, o2->ipIfStatsHCInOctets);
11376     UPDATE_MIB(o1, ipIfStatsHCInForwDatagrams,
11377         o2->ipIfStatsHCInForwDatagrams);
11378     UPDATE_MIB(o1, ipIfStatsHCInDelivers, o2->ipIfStatsHCInDelivers);
11379     UPDATE_MIB(o1, ipIfStatsHCOutRequests, o2->ipIfStatsHCOutRequests);
11380     UPDATE_MIB(o1, ipIfStatsHCOutForwDatagrams,
11381         o2->ipIfStatsHCOutForwDatagrams);
11382     UPDATE_MIB(o1, ipIfStatsOutFragReqds, o2->ipIfStatsOutFragReqds);
11383     UPDATE_MIB(o1, ipIfStatsHCOutTransmits, o2->ipIfStatsHCOutTransmits);
11384     UPDATE_MIB(o1, ipIfStatsHCOutOctets, o2->ipIfStatsHCOutOctets);
11385     UPDATE_MIB(o1, ipIfStatsHCInMcastPkts, o2->ipIfStatsHCInMcastPkts);
11386     UPDATE_MIB(o1, ipIfStatsHCInMcastOctets, o2->ipIfStatsHCInMcastOctets);
11387     UPDATE_MIB(o1, ipIfStatsHCOutMcastPkts, o2->ipIfStatsHCOutMcastPkts);
11388     UPDATE_MIB(o1, ipIfStatsHCOutMcastOctets,
11389         o2->ipIfStatsHCOutMcastOctets);
11390     UPDATE_MIB(o1, ipIfStatsHCInBcastPkts, o2->ipIfStatsHCInBcastPkts);
11391     UPDATE_MIB(o1, ipIfStatsHCOutBcastPkts, o2->ipIfStatsHCOutBcastPkts);
11392     UPDATE_MIB(o1, ipsecInSucceeded, o2->ipsecInSucceeded);
11393     UPDATE_MIB(o1, ipsecInFailed, o2->ipsecInFailed);
11394     UPDATE_MIB(o1, ipInCksumErrs, o2->ipInCksumErrs);
11395     UPDATE_MIB(o1, tcpInErrs, o2->tcpInErrs);
11396     UPDATE_MIB(o1, udpNoPorts, o2->udpNoPorts);
11397 }

11399 void
11400 ip_mib2_add_icmp6_stats(mib2_ipv6IfIcmpEntry_t *o1, mib2_ipv6IfIcmpEntry_t *o2)
11401 {
11402     UPDATE_MIB(o1, ipv6IfIcmpInMsgs, o2->ipv6IfIcmpInMsgs);
11403     UPDATE_MIB(o1, ipv6IfIcmpInErrors, o2->ipv6IfIcmpInErrors);
11404     UPDATE_MIB(o1, ipv6IfIcmpInDestUnreachs, o2->ipv6IfIcmpInDestUnreachs);
11405     UPDATE_MIB(o1, ipv6IfIcmpInAdminProhibs, o2->ipv6IfIcmpInAdminProhibs);
11406     UPDATE_MIB(o1, ipv6IfIcmpInTimeExcds, o2->ipv6IfIcmpInTimeExcds);
11407     UPDATE_MIB(o1, ipv6IfIcmpInParmProblems, o2->ipv6IfIcmpInParmProblems);
11408     UPDATE_MIB(o1, ipv6IfIcmpInPktTooBigs, o2->ipv6IfIcmpInPktTooBigs);
11409     UPDATE_MIB(o1, ipv6IfIcmpInEchos, o2->ipv6IfIcmpInEchos);
11410     UPDATE_MIB(o1, ipv6IfIcmpInEchoReplies, o2->ipv6IfIcmpInEchoReplies);
11411     UPDATE_MIB(o1, ipv6IfIcmpInRouterSolicits,
11412         o2->ipv6IfIcmpInRouterSolicits);
11413     UPDATE_MIB(o1, ipv6IfIcmpInRouterAdvertisements,

```

```

11414         o2->ipv6IfIcmpInRouterAdvertisements);
11415     UPDATE_MIB(o1, ipv6IfIcmpInNeighborSolicits,
11416         o2->ipv6IfIcmpInNeighborSolicits);
11417     UPDATE_MIB(o1, ipv6IfIcmpInNeighborAdvertisements,
11418         o2->ipv6IfIcmpInNeighborAdvertisements);
11419     UPDATE_MIB(o1, ipv6IfIcmpInRedirects, o2->ipv6IfIcmpInRedirects);
11420     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembQueries,
11421         o2->ipv6IfIcmpInGroupMembQueries);
11422     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembResponses,
11423         o2->ipv6IfIcmpInGroupMembResponses);
11424     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembReductions,
11425         o2->ipv6IfIcmpInGroupMembReductions);
11426     UPDATE_MIB(o1, ipv6IfIcmpOutMsgs, o2->ipv6IfIcmpOutMsgs);
11427     UPDATE_MIB(o1, ipv6IfIcmpOutErrors, o2->ipv6IfIcmpOutErrors);
11428     UPDATE_MIB(o1, ipv6IfIcmpOutDestUnreachs,
11429         o2->ipv6IfIcmpOutDestUnreachs);
11430     UPDATE_MIB(o1, ipv6IfIcmpOutAdminProhibs,
11431         o2->ipv6IfIcmpOutAdminProhibs);
11432     UPDATE_MIB(o1, ipv6IfIcmpOutTimeExcds, o2->ipv6IfIcmpOutTimeExcds);
11433     UPDATE_MIB(o1, ipv6IfIcmpOutParmProblems,
11434         o2->ipv6IfIcmpOutParmProblems);
11435     UPDATE_MIB(o1, ipv6IfIcmpOutPktTooBigs, o2->ipv6IfIcmpOutPktTooBigs);
11436     UPDATE_MIB(o1, ipv6IfIcmpOutEchos, o2->ipv6IfIcmpOutEchos);
11437     UPDATE_MIB(o1, ipv6IfIcmpOutEchoReplies, o2->ipv6IfIcmpOutEchoReplies);
11438     UPDATE_MIB(o1, ipv6IfIcmpOutRouterSolicits,
11439         o2->ipv6IfIcmpOutRouterSolicits);
11440     UPDATE_MIB(o1, ipv6IfIcmpOutRouterAdvertisements,
11441         o2->ipv6IfIcmpOutRouterAdvertisements);
11442     UPDATE_MIB(o1, ipv6IfIcmpOutNeighborSolicits,
11443         o2->ipv6IfIcmpOutNeighborSolicits);
11444     UPDATE_MIB(o1, ipv6IfIcmpOutNeighborAdvertisements,
11445         o2->ipv6IfIcmpOutNeighborAdvertisements);
11446     UPDATE_MIB(o1, ipv6IfIcmpOutRedirects, o2->ipv6IfIcmpOutRedirects);
11447     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembQueries,
11448         o2->ipv6IfIcmpOutGroupMembQueries);
11449     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembResponses,
11450         o2->ipv6IfIcmpOutGroupMembResponses);
11451     UPDATE_MIB(o1, ipv6IfIcmpOutGroupMembReductions,
11452         o2->ipv6IfIcmpOutGroupMembReductions);
11453     UPDATE_MIB(o1, ipv6IfIcmpInOverflows, o2->ipv6IfIcmpInOverflows);
11454     UPDATE_MIB(o1, ipv6IfIcmpBadHoplimit, o2->ipv6IfIcmpBadHoplimit);
11455     UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborAdvertisements,
11456         o2->ipv6IfIcmpInBadNeighborAdvertisements);
11457     UPDATE_MIB(o1, ipv6IfIcmpInBadNeighborSolicitations,
11458         o2->ipv6IfIcmpInBadNeighborSolicitations);
11459     UPDATE_MIB(o1, ipv6IfIcmpInBadRedirects, o2->ipv6IfIcmpInBadRedirects);
11460     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembTotal,
11461         o2->ipv6IfIcmpInGroupMembTotal);
11462     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadQueries,
11463         o2->ipv6IfIcmpInGroupMembBadQueries);
11464     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembBadReports,
11465         o2->ipv6IfIcmpInGroupMembBadReports);
11466     UPDATE_MIB(o1, ipv6IfIcmpInGroupMembOurReports,
11467         o2->ipv6IfIcmpInGroupMembOurReports);
11468 }

11470 /*
11471  * Called before the options are updated to check if this packet will
11472  * be source routed from here.
11473  * This routine assumes that the options are well formed i.e. that they
11474  * have already been checked.
11475  */
11476 boolean_t
11477 ip_source_routed(ipha_t *ipha, ip_stack_t *ipst)
11478 {
11479     ipoptp_t      opts;

```



```

11480     uchar_t      *opt;
11481     uint8_t      optval;
11482     uint8_t      optlen;
11483     ipaddr_t      dst;

11485     if (IS_SIMPLE_IPH(ipha)) {
11486         ip2dbg(("not source routed\n"));
11487         return (B_FALSE);
11488     }
11489     dst = ipha->ipha_dst;
11490     for (optval = ipoctp_first(&opts, ipha);
11491          optval != IPOPT_EOL;
11492          optval = ipoctp_next(&opts)) {
11493         ASSERT((opts.ipoctp_flags & IPOPT_ERROR) == 0);
11494         opt = opts.ipoctp_cur;
11495         optlen = opts.ipoctp_len;
11496         ip2dbg(("ip_source_routed: opt %d, len %d\n",
11497                optval, optlen));
11498         switch (optval) {
11499             uint32_t off;
11500             case IPOPT_SSRR:
11501             case IPOPT_LSRR:
11502                 /*
11503                  * If dst is one of our addresses and there are some
11504                  * entries left in the source route return (true).
11505                  */
11506                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11507                     ip2dbg(("ip_source_routed: not next"
11508                            " source route 0x%x\n",
11509                            ntohl(dst)));
11510                     return (B_FALSE);
11511                 }
11512                 off = opt[IPOPT_OFFSET];
11513                 off--;
11514                 if (optlen < IP_ADDR_LEN ||
11515                     off > optlen - IP_ADDR_LEN) {
11516                     /* End of source route */
11517                     ip1dbg(("ip_source_routed: end of SR\n"));
11518                     return (B_FALSE);
11519                 }
11520                 return (B_TRUE);
11521             }
11522     }
11523     ip2dbg(("not source routed\n"));
11524     return (B_FALSE);
11525 }

11527 /*
11528  * ip_unbind is called by the transports to remove a conn from
11529  * the fanout table.
11530  */
11531 void
11532 ip_unbind(conn_t *connp)
11533 {
11535     ASSERT(!MUTEX_HELD(&connp->conn_lock));

11537     if (is_system_labeled() && connp->conn_anon_port) {
11538         (void) tsol_mlp_anon(crgetzone(connp->conn_cred),
11539                             connp->conn_mlp_type, connp->conn_proto,
11540                             ntohs(connp->conn_lport), B_FALSE);
11541         connp->conn_anon_port = 0;
11542     }
11543     connp->conn_mlp_type = mlptSingle;
11545     ipcl_hash_remove(connp);

```

```

11546 }

11548 /*
11549  * Used for deciding the MSS size for the upper layer. Thus
11550  * we need to check the outbound policy values in the conn.
11551  */
11552 int
11553 conn_ipsec_length(conn_t *connp)
11554 {
11555     ipsec_latch_t *ipl;

11557     ipl = connp->conn_latch;
11558     if (ipl == NULL)
11559         return (0);

11561     if (connp->conn_ixa->ixa_ipsec_policy == NULL)
11562         return (0);

11564     return (connp->conn_ixa->ixa_ipsec_policy->ipsp_act->ipa_ovhd);
11565 }

11567 /*
11568  * Returns an estimate of the IPsec headers size. This is used if
11569  * we don't want to call into IPsec to get the exact size.
11570  */
11571 int
11572 ipsec_out_extra_length(ip_xmit_attr_t *ixa)
11573 {
11574     ipsec_action_t *a;

11576     if (!(ixa->ixa_flags & IXAF_IPSEC_SECURE))
11577         return (0);

11579     a = ix->ixa_ipsec_action;
11580     if (a == NULL) {
11581         ASSERT(ixa->ixa_ipsec_policy != NULL);
11582         a = ix->ixa_ipsec_policy->ipsp_act;
11583     }
11584     ASSERT(a != NULL);

11586     return (a->ipa_ovhd);
11587 }

11589 /*
11590  * If there are any source route options, return the true final
11591  * destination. Otherwise, return the destination.
11592  */
11593 ipaddr_t
11594 ip_get_dst(ipha_t *ipha)
11595 {
11596     ipoctp_t      opts;
11597     uchar_t      *opt;
11598     uint8_t      optval;
11599     uint8_t      optlen;
11600     ipaddr_t      dst;
11601     uint32_t off;

11603     dst = ipha->ipha_dst;

11605     if (IS_SIMPLE_IPH(ipha))
11606         return (dst);

11608     for (optval = ipoctp_first(&opts, ipha);
11609          optval != IPOPT_EOL;
11610          optval = ipoctp_next(&opts)) {
11611         opt = opts.ipoctp_cur;

```

```

11612     optlen = opts.ipoptp_len;
11613     ASSERT((opts.ipoptp_flags & IPOPT_ERROR) == 0);
11614     switch (optval) {
11615     case IPOPT_SSRR:
11616     case IPOPT_LSRR:
11617         off = opt[IPOPT_OFFSET];
11618         /*
11619          * If one of the conditions is true, it means
11620          * end of options and dst already has the right
11621          * value.
11622          */
11623         if (!(optlen < IP_ADDR_LEN || off > optlen - 3)) {
11624             off = optlen - IP_ADDR_LEN;
11625             bcopy(&opt[off], &dst, IP_ADDR_LEN);
11626         }
11627         return (dst);
11628     default:
11629         break;
11630     }
11631 }

11633     return (dst);
11634 }

11636 /*
11637  * Outbound IP fragmentation routine.
11638  * Assumes the caller has checked whether or not fragmentation should
11639  * be allowed. Here we copy the DF bit from the header to all the generated
11640  * fragments.
11641  */
11642 int
11643 ip_fragment_v4(mblk_t *mp_orig, nce_t *nce, iaflags_t ixaflags,
11644               uint_t pkt_len, uint32_t max_frag, uint32_t xmit_hint, zoneid_t szone,
11645               zoneid_t nolzid, pfirepostfrag_t postfragfn, uintptr_t *ixa_cookie)
11646 {
11647     int             il;
11648     int             hdr_len;
11649     mblk_t          *hdr_mp;
11650     ipha_t          *ipha;
11651     int             ip_data_end;
11652     int             len;
11653     mblk_t          *mp = mp_orig;
11654     int             offset;
11655     ill_t           *ill = nce->nce_ill;
11656     ip_stack_t      *ipst = ill->ill_ipst;
11657     mblk_t          *carve_mp;
11658     uint32_t         frag_flag;
11659     uint_t          priority = mp->b_band;
11660     int             error = 0;

11662     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragReqds);

11664     if (pkt_len != msgdsize(mp)) {
11665         ip0dbg(("Packet length mismatch: %d, %ld\n",
11666              pkt_len, msgdsize(mp)));
11667         freemsg(mp);
11668         return (EINVAL);
11669     }

11671     if (max_frag == 0) {
11672         ip1dbg(("ip_fragment_v4: max_frag is zero. Dropping packet\n"));
11673         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11674         ip_drop_output("FragFails: zero max_frag", mp, ill);
11675         freemsg(mp);
11676         return (EINVAL);
11677     }

```

```

11679     ASSERT(MBLKL(mp) >= sizeof (ipha_t));
11680     ipha = (ipha_t *)mp->b_rptr;
11681     ASSERT(ntohs(ipha->ipha_length) == pkt_len);
11682     frag_flag = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_DF;

11684     /*
11685      * Establish the starting offset. May not be zero if we are fragging
11686      * a fragment that is being forwarded.
11687      */
11688     offset = ntohs(ipha->ipha_fragment_offset_and_flags) & IPH_OFFSET;

11690     /* TODO why is this test needed? */
11691     if (((max_frag - ntohs(ipha->ipha_length)) & ~7) < 8) {
11692         /* TODO: notify ulp somehow */
11693         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11694         ip_drop_output("FragFails: bad starting offset", mp, ill);
11695         freemsg(mp);
11696         return (EINVAL);
11697     }

11699     hdr_len = IPH_HDR_LENGTH(ipha);
11700     ipha->ipha_hdr_checksum = 0;

11702     /*
11703      * Establish the number of bytes maximum per frag, after putting
11704      * in the header.
11705      */
11706     len = (max_frag - hdr_len) & ~7;

11708     /* Get a copy of the header for the trailing frags */
11709     hdr_mp = ip_fragment_copyhdr((uchar_t *)ipha, hdr_len, offset, ipst,
11710                                mp);
11711     if (hdr_mp == NULL) {
11712         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11713         ip_drop_output("FragFails: no hdr_mp", mp, ill);
11714         freemsg(mp);
11715         return (ENOBUFS);
11716     }

11718     /* Store the starting offset, with the MoreFrag flag. */
11719     il = offset | IPH_MF | frag_flag;
11720     ipha->ipha_fragment_offset_and_flags = htons((uint16_t)il);

11722     /* Establish the ending byte offset, based on the starting offset. */
11723     offset <= 3;
11724     ip_data_end = offset + ntohs(ipha->ipha_length) - hdr_len;

11726     /* Store the length of the first fragment in the IP header. */
11727     il = len + hdr_len;
11728     ASSERT(il <= IP_MAXPACKET);
11729     ipha->ipha_length = htons((uint16_t)il);

11731     /*
11732      * Compute the IP header checksum for the first frag. We have to
11733      * watch out that we stop at the end of the header.
11734      */
11735     ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

11737     /*
11738      * Now carve off the first frag. Note that this will include the
11739      * original IP header.
11740      */
11741     if (! (mp = ip_carve_mp(&mp_orig, il))) {
11742         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11743         ip_drop_output("FragFails: could not carve mp", mp_orig, ill);

```

```

11744         freeb(hdr_mp);
11745         freemsg(mp_orig);
11746         return (ENOBUFS);
11747     }
11749     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);
11751     error = postfragfn(mp, nce, ixaflags, il, xmit_hint, szone, nolzid,
11752         ixa_cookie);
11753     if (error != 0 && error != EWOULDBLOCK) {
11754         /* No point in sending the other fragments */
11755         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11756         ip_drop_output("FragFails: postfragfn failed", mp_orig, ill);
11757         freeb(hdr_mp);
11758         freemsg(mp_orig);
11759         return (error);
11760     }
11762     /* No need to redo state machine in loop */
11763     ixaflags &= ~IXAF_REACH_CONF;
11765     /* Advance the offset to the second frag starting point. */
11766     offset += len;
11767     /*
11768      * Update hdr_len from the copied header - there might be less options
11769      * in the later fragments.
11770      */
11771     hdr_len = IPH_HDR_LENGTH(hdr_mp->b_rptr);
11772     /* Loop until done. */
11773     for (;;) {
11774         uint16_t    offset_and_flags;
11775         uint16_t    ip_len;
11777         if (ip_data_end - offset > len) {
11778             /*
11779              * Carve off the appropriate amount from the original
11780              * datagram.
11781              */
11782             if (!(carve_mp = ip_carve_mp(&mp_orig, len))) {
11783                 mp = NULL;
11784                 break;
11785             }
11786             /*
11787              * More frags after this one. Get another copy
11788              * of the header.
11789              */
11790             if (carve_mp->b_datap->db_ref == 1 &&
11791                 hdr_mp->b_wptr - hdr_mp->b_rptr <
11792                 carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11793                 /* Inline IP header */
11794                 carve_mp->b_rptr -= hdr_mp->b_wptr -
11795                     hdr_mp->b_rptr;
11796                 bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11797                     hdr_mp->b_wptr - hdr_mp->b_rptr);
11798                 mp = carve_mp;
11799             } else {
11800                 if (!(mp = copyb(hdr_mp))) {
11801                     freemsg(carve_mp);
11802                     break;
11803                 }
11804                 /* Get priority marking, if any. */
11805                 mp->b_band = priority;
11806                 mp->b_cont = carve_mp;
11807             }
11808             ipha = (ipha_t *)mp->b_rptr;
11809             offset_and_flags = IPH_MF;

```

```

11810         } else {
11811             /*
11812              * Last frag. Consume the header. Set len to
11813              * the length of this last piece.
11814              */
11815             len = ip_data_end - offset;
11817             /*
11818              * Carve off the appropriate amount from the original
11819              * datagram.
11820              */
11821             if (!(carve_mp = ip_carve_mp(&mp_orig, len))) {
11822                 mp = NULL;
11823                 break;
11824             }
11825             if (carve_mp->b_datap->db_ref == 1 &&
11826                 hdr_mp->b_wptr - hdr_mp->b_rptr <
11827                 carve_mp->b_rptr - carve_mp->b_datap->db_base) {
11828                 /* Inline IP header */
11829                 carve_mp->b_rptr -= hdr_mp->b_wptr -
11830                     hdr_mp->b_rptr;
11831                 bcopy(hdr_mp->b_rptr, carve_mp->b_rptr,
11832                     hdr_mp->b_wptr - hdr_mp->b_rptr);
11833                 mp = carve_mp;
11834                 freeb(hdr_mp);
11835                 hdr_mp = mp;
11836             } else {
11837                 mp = hdr_mp;
11838                 /* Get priority marking, if any. */
11839                 mp->b_band = priority;
11840                 mp->b_cont = carve_mp;
11841             }
11842             ipha = (ipha_t *)mp->b_rptr;
11843             /* A frag of a frag might have IPH_MF non-zero */
11844             offset_and_flags =
11845                 ntohs(ipha->ipha_fragment_offset_and_flags) &
11846                 IPH_MF;
11847         }
11848         offset_and_flags |= (uint16_t)(offset >> 3);
11849         offset_and_flags |= (uint16_t)frag_flag;
11850         /* Store the offset and flags in the IP header. */
11851         ipha->ipha_fragment_offset_and_flags = htons(offset_and_flags);
11853         /* Store the length in the IP header. */
11854         ip_len = (uint16_t)(len + hdr_len);
11855         ipha->ipha_length = htons(ip_len);
11857         /*
11858          * Set the IP header checksum. Note that mp is just
11859          * the header, so this is easy to pass to ip_csum.
11860          */
11861         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
11863         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragCreates);
11865         error = postfragfn(mp, nce, ixaflags, ip_len, xmit_hint, szone,
11866             nolzid, ixa_cookie);
11867         /* All done if we just consumed the hdr_mp. */
11868         if (mp == hdr_mp) {
11869             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragOKs);
11870             return (error);
11871         }
11872         if (error != 0 && error != EWOULDBLOCK) {
11873             DTRACE_PROBE2(ip_xmit_frag_fail, ill_t *, ill,
11874                 mblk_t *, hdr_mp);
11875             /* No point in sending the other fragments */

```

```

11876         break;
11877     }

11879     /* Otherwise, advance and loop. */
11880     offset += len;
11881 }
11882 /* Clean up following allocation failure. */
11883 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutFragFails);
11884 ip_drop_output("FragFails: loop ended", NULL, ill);
11885 if (mp != hdr_mp)
11886     freeb(hdr_mp);
11887 if (mp != mp_orig)
11888     freemsg(mp_orig);
11889 return (error);
11890 }

11892 /*
11893  * Copy the header plus those options which have the copy bit set
11894  */
11895 static mblk_t *
11896 ip_fragment_copyhdr(uchar_t *rptr, int hdr_len, int offset, ip_stack_t *ipst,
11897     mblk_t *src)
11898 {
11899     mblk_t *mp;
11900     uchar_t *up;

11902     /*
11903      * Quick check if we need to look for options without the copy bit
11904      * set
11905      */
11906     mp = allocb_tmpl(ipst->ips_ip_wroff_extra + hdr_len, src);
11907     if (!mp)
11908         return (mp);
11909     mp->b_rptr += ipst->ips_ip_wroff_extra;
11910     if (hdr_len == IP_SIMPLE_HDR_LENGTH || offset != 0) {
11911         bcopy(rptr, mp->b_rptr, hdr_len);
11912         mp->b_wptr += hdr_len + ipst->ips_ip_wroff_extra;
11913         return (mp);
11914     }
11915     up = mp->b_rptr;
11916     bcopy(rptr, up, IP_SIMPLE_HDR_LENGTH);
11917     up += IP_SIMPLE_HDR_LENGTH;
11918     rptr += IP_SIMPLE_HDR_LENGTH;
11919     hdr_len -= IP_SIMPLE_HDR_LENGTH;
11920     while (hdr_len > 0) {
11921         uint32_t optval;
11922         uint32_t optlen;

11924         optval = *rptr;
11925         if (optval == IPOPT_EOL)
11926             break;
11927         if (optval == IPOPT_NOP)
11928             optlen = 1;
11929         else
11930             optlen = rptr[1];
11931         if (optval & IPOPT_COPY) {
11932             bcopy(rptr, up, optlen);
11933             up += optlen;
11934         }
11935         rptr += optlen;
11936         hdr_len -= optlen;
11937     }
11938     /*
11939      * Make sure that we drop an even number of words by filling
11940      * with EOL to the next word boundary.
11941      */

```

```

11942     for (hdr_len = up - (mp->b_rptr + IP_SIMPLE_HDR_LENGTH);
11943          hdr_len & 0x3; hdr_len++)
11944         *up++ = IPOPT_EOL;
11945     mp->b_wptr = up;
11946     /* Update header length */
11947     mp->b_rptr[0] = (uint8_t)((IP_VERSION << 4) | ((up - mp->b_rptr) >> 2));
11948     return (mp);
11949 }

11951 /*
11952  * Update any source route, record route, or timestamp options when
11953  * sending a packet back to ourselves.
11954  * Check that we are at end of strict source route.
11955  * The options have been sanity checked by ip_output_options().
11956  */
11957 void
11958 ip_output_local_options(ipha_t *ipha, ip_stack_t *ipst)
11959 {
11960     ipoptp_t    opts;
11961     uchar_t     *opt;
11962     uint8_t     optval;
11963     uint8_t     optlen;
11964     ipaddr_t    dst;
11965     uint32_t    ts;
11966     timestruc_t now;

11968     for (optval = ipoptp_first(&opts, ipha);
11969          optval != IPOPT_EOL;
11970          optval = ipoptp_next(&opts)) {
11971         opt = opts.ipoptp_cur;
11972         optlen = opts.ipoptp_len;
11973         ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11974         switch (optval) {
11975             uint32_t off;
11976             case IPOPT_SSRR:
11977             case IPOPT_LSRR:
11978                 off = opt[IPOPT_OFFSET];
11979                 off--;
11980                 if (optlen < IP_ADDR_LEN ||
11981                     off > optlen - IP_ADDR_LEN) {
11982                     /* End of source route */
11983                     break;
11984                 }
11985             /*
11986              * This will only happen if two consecutive entries
11987              * in the source route contains our address or if
11988              * it is a packet with a loose source route which
11989              * reaches us before consuming the whole source route
11990              */

11992             if (optval == IPOPT_SSRR) {
11993                 return;
11994             }
11995             /*
11996              * Hack: instead of dropping the packet truncate the
11997              * source route to what has been used by filling the
11998              * rest with IPOPT_NOP.
11999              */
12000             opt[IPOPT_OLEN] = (uint8_t)off;
12001             while (off < optlen) {
12002                 opt[off++] = IPOPT_NOP;
12003             }
12004             break;
12005             case IPOPT_RR:
12006                 off = opt[IPOPT_OFFSET];
12007                 off--;

```

```

12008         if (optlen < IP_ADDR_LEN ||
12009             off > optlen - IP_ADDR_LEN) {
12010             /* No more room - ignore */
12011             ipldbg(("
12012                 ip_output_local_options: end of RR\n"));
12013             break;
12014         }
12015         dst = htonl(INADDR_LOOPBACK);
12016         bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
12017         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
12018         break;
12019     case IPOPT_TS:
12020         /* Insert timestamp if there is room */
12021         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
12022             case IPOPT_TS_TSONLY:
12023                 off = IPOPT_TS_TIMELEN;
12024                 break;
12025             case IPOPT_TS_PRESPEC:
12026             case IPOPT_TS_PRESPEC_RFC791:
12027                 /* Verify that the address matched */
12028                 off = opt[IPOPT_OFFSET] - 1;
12029                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
12030                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
12031                     /* Not for us */
12032                     break;
12033                 }
12034                 /* FALLTHRU */
12035             case IPOPT_TS_TSANDADDR:
12036                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
12037                 break;
12038             default:
12039                 /*
12040                  * ip_*put_options should have already
12041                  * dropped this packet.
12042                  */
12043                 cmn_err(CE_PANIC, "ip_output_local_options: "
12044                     "unknown IT - bug in ip_output_options?\n");
12045                 return; /* Keep "lint" happy */
12046             }
12047         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
12048             /* Increase overflow counter */
12049             off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
12050             opt[IPOPT_POS_OV_FLG] = (uint8_t)
12051                 (opt[IPOPT_POS_OV_FLG] & 0x0F) |
12052                 (off << 4);
12053             break;
12054         }
12055         off = opt[IPOPT_OFFSET] - 1;
12056         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
12057             case IPOPT_TS_PRESPEC:
12058             case IPOPT_TS_PRESPEC_RFC791:
12059             case IPOPT_TS_TSANDADDR:
12060                 dst = htonl(INADDR_LOOPBACK);
12061                 bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
12062                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
12063                 /* FALLTHRU */
12064             case IPOPT_TS_TSONLY:
12065                 off = opt[IPOPT_OFFSET] - 1;
12066                 /* Compute # of milliseconds since midnight */
12067                 gethrtime(&now);
12068                 ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
12069                     now.tv_nsec / (NANOSEC / MILLISEC);
12070                 bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
12071                 opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
12072                 break;
12073             }

```

```

12074             break;
12075         }
12076     }
12077 }

12079 /*
12080  * Prepend an M_DATA fastpath header, and if none present prepend a
12081  * DL_UNITDATA_REQ. Frees the mblk on failure.
12082  *
12083  * nce_dlur_mp and nce_fp_mp can not disappear once they have been set.
12084  * If there is a change to them, the nce will be deleted (condemned) and
12085  * a new nce_t will be created when packets are sent. Thus we need no locks
12086  * to access those fields.
12087  *
12088  * We preserve b_band to support IPQoS. If a DL_UNITDATA_REQ is prepended
12089  * we place b_band in dl_priority.dl_max.
12090  */
12091 static mblk_t *
12092 ip_xmit_attach_llhdr(mblk_t *mp, nce_t *nce)
12093 {
12094     uint_t hlen;
12095     mblk_t *mpl;
12096     uint_t priority;
12097     uchar_t *rptr;

12099     rptr = mp->b_rptr;

12101     ASSERT(DB_TYPE(mp) == M_DATA);
12102     priority = mp->b_band;

12104     ASSERT(nce != NULL);
12105     if ((mpl = nce->nce_fp_mp) != NULL) {
12106         hlen = MBLKL(mpl);
12107         /*
12108          * Check if we have enough room to prepend fastpath
12109          * header
12110          */
12111         if (hlen != 0 && (rptr - mp->b_datap->db_base) >= hlen) {
12112             rptr -= hlen;
12113             bcopy(mpl->b_rptr, rptr, hlen);
12114             /*
12115              * Set the b_rptr to the start of the link layer
12116              * header
12117              */
12118             mp->b_rptr = rptr;
12119             return (mp);
12120         }
12121         mpl = copyb(mpl);
12122         if (mpl == NULL) {
12123             ill_t *ill = nce->nce_ill;

12125             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12126             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12127             freemsg(mp);
12128             return (NULL);
12129         }
12130         mpl->b_band = priority;
12131         mpl->b_cont = mp;
12132         DB_CKSUMSTART(mpl) = DB_CKSUMSTART(mp);
12133         DB_CKSUMSTUFF(mpl) = DB_CKSUMSTUFF(mp);
12134         DB_CKSUMEND(mpl) = DB_CKSUMEND(mp);
12135         DB_CKSUMFLAGS(mpl) = DB_CKSUMFLAGS(mp);
12136         DB_LSOMSS(mpl) = DB_LSOMSS(mp);
12137         DTRACE_PROBE1(ip_xmit_copyb, (mblk_t *), mpl);
12138         /*
12139          * XXX disable ICK_VALID and compute checksum

```

```

12140         * here; can happen if nce_fp_mp changes and
12141         * it can't be copied now due to insufficient
12142         * space. (unlikely, fp mp can change, but it
12143         * does not increase in length)
12144         */
12145         return (mpl);
12146     }
12147     mpl = copyb(nce->nce_dlur_mp);

12149     if (mpl == NULL) {
12150         ill_t *ill = nce->nce_ill;

12152         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12153         ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12154         freemsg(mp);
12155         return (NULL);
12156     }
12157     mpl->b_cont = mp;
12158     if (priority != 0) {
12159         mpl->b_band = priority;
12160         ((dl_unitdata_req_t *) (mpl->b_rptr))->dl_priority.dl_max =
12161             priority;
12162     }
12163     return (mpl);
12164 #undef rp_ptr
12165 }

12167 /*
12168  * Finish the outbound IPsec processing. This function is called from
12169  * ipsec_out_process() if the IPsec packet was processed
12170  * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12171  * asynchronously.
12172  *
12173  * This is common to IPv4 and IPv6.
12174  */
12175 int
12176 ip_output_post_ipsec(mblk_t *mp, ip_xmit_attr_t *ixa)
12177 {
12178     iaflags_t      iaxaflags = ixa->ixa_flags;
12179     uint_t          pktlen;

12182     /* AH/ESP don't update ixa_pktlen when they modify the packet */
12183     if (iaflags & IXAF_IS_IPV4) {
12184         ipha_t      *ipha = (ipha_t *) mp->b_rptr;

12186         ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);
12187         pktlen = ntohs(ipha->ipha_length);
12188     } else {
12189         ip6_t      *ip6h = (ip6_t *) mp->b_rptr;

12191         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12192         pktlen = ntohs(ip6h->ip6_plen) + IPV6_HDR_LEN;
12193     }

12195     /*
12196      * We release any hard reference on the SAs here to make
12197      * sure the SAs can be garbage collected. ipsr_sa has a soft reference
12198      * on the SAs.
12199      * If in the future we want the hard latching of the SAs in the
12200      * ip_xmit_attr_t then we should remove this.
12201      */
12202     if (ixa->ixa_ipsec_esp_sa != NULL) {
12203         IPSA_REFRELE(ixa->ixa_ipsec_esp_sa);
12204         ixa->ixa_ipsec_esp_sa = NULL;
12205     }

```

```

12206     if (ixa->ixa_ipsec_ah_sa != NULL) {
12207         IPSA_REFRELE(ixa->ixa_ipsec_ah_sa);
12208         ixa->ixa_ipsec_ah_sa = NULL;
12209     }

12211     /* Do we need to fragment? */
12212     if ((ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR) ||
12213         pktlen > ixa->ixa_fragsize) {
12214         if (iaflags & IXAF_IS_IPV4) {
12215             ASSERT(!(ixa->ixa_flags & IXAF_IPV6_ADD_FRAGHDR));
12216             /*
12217              * We check for the DF case in ipsec_out_process
12218              * hence this only handles the non-DF case.
12219              */
12220             return (ip_fragment_v4(mp, ixa->ixa_nce, ixa->ixa_flags,
12221                 pktlen, ixa->ixa_fragsize,
12222                 ixa->ixa_xmit_hint, ixa->ixa_zoneid,
12223                 ixa->ixa_no_loop_zoneid, ixa->ixa_postfragfn,
12224                 &ixa->ixa_cookie));
12225         } else {
12226             mp = ip_fraghdr_add_v6(mp, ixa->ixa_ident, ixa);
12227             if (mp == NULL) {
12228                 /* MIB and ip_drop_output already done */
12229                 return (ENOMEM);
12230             }
12231             pktlen += sizeof(ip6_frag_t);
12232             if (pktlen > ixa->ixa_fragsize) {
12233                 return (ip_fragment_v6(mp, ixa->ixa_nce,
12234                     ixa->ixa_flags, pktlen,
12235                     ixa->ixa_fragsize, ixa->ixa_xmit_hint,
12236                     ixa->ixa_zoneid, ixa->ixa_no_loop_zoneid,
12237                     ixa->ixa_postfragfn, &ixa->ixa_cookie));
12238             }
12239         }
12240     }
12241     return ((ixa->ixa_postfragfn)(mp, ixa->ixa_nce, ixa->ixa_flags,
12242         pktlen, ixa->ixa_xmit_hint, ixa->ixa_zoneid,
12243         ixa->ixa_no_loop_zoneid, NULL));
12244 }

12246 /*
12247  * Finish the inbound IPsec processing. This function is called from
12248  * ipsec_out_process() if the IPsec packet was processed
12249  * synchronously, or from {ah,esp}_kcf_callback_outbound() if it was processed
12250  * asynchronously.
12251  *
12252  * This is common to IPv4 and IPv6.
12253  */
12254 void
12255 ip_input_post_ipsec(mblk_t *mp, ip_recv_attr_t *ira)
12256 {
12257     iaflags_t      iraflags = ira->ira_flags;

12259     /* Length might have changed */
12260     if (iraflags & IRAF_IS_IPV4) {
12261         ipha_t      *ipha = (ipha_t *) mp->b_rptr;

12263         ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);
12264         ira->ira_pktlen = ntohs(ipha->ipha_length);
12265         ira->ira_ip_hdr_length = IPH_HDR_LENGTH(ipha);
12266         ira->ira_protocol = ipha->ipha_protocol;

12268         ip_fanout_v4(mp, ipha, ira);
12269     } else {
12270         ip6_t      *ip6h = (ip6_t *) mp->b_rptr;
12271         uint8_t      *nextthdrp;

```

```

12273     ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);
12274     ira->ira_pktlen = ntohs(ip6h->ip6_plen) + IPV6_HDR_LEN;
12275     if (!ip_hdr_length Nexthdr_v6(mp, ip6h, &ira->ira_ip_hdr_length,
12276         &nexthdrp)) {
12277         /* Malformed packet */
12278         BUMP_MIB(ira->ira_ill->ill_ip_mib, ipIfStatsInDiscards);
12279         ip_drop_input("ipIfStatsInDiscards", mp, ira->ira_ill);
12280         freemsg(mp);
12281         return;
12282     }
12283     ira->ira_protocol = *nexthdrp;
12284     ip_fanout_v6(mp, ip6h, ira);
12285 }
12286 }

12288 /*
12289  * Select which AH & ESP SA's to use (if any) for the outbound packet.
12290  *
12291  * If this function returns B_TRUE, the requested SA's have been filled
12292  * into the ixa_ipsec_*_sa pointers.
12293  *
12294  * If the function returns B_FALSE, the packet has been "consumed", most
12295  * likely by an ACQUIRE sent up via PF_KEY to a key management daemon.
12296  *
12297  * The SA references created by the protocol-specific "select"
12298  * function will be released in ip_output_post_ipsec.
12299  */
12300 static boolean_t
12301 ipsec_out_select_sa(mblk_t *mp, ip_xmit_attr_t *ixa)
12302 {
12303     boolean_t need_ah_acquire = B_FALSE, need_esp_acquire = B_FALSE;
12304     ipsec_policy_t *pp;
12305     ipsec_action_t *ap;

12307     ASSERT(ixa->ixa_flags & IXAF_IPSEC_SECURE);
12308     ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12309         (ixa->ixa_ipsec_action != NULL));

12311     ap = ixa->ixa_ipsec_action;
12312     if (ap == NULL) {
12313         pp = ixa->ixa_ipsec_policy;
12314         ASSERT(pp != NULL);
12315         ap = pp->ipsp_act;
12316         ASSERT(ap != NULL);
12317     }

12319     /*
12320      * We have an action. now, let's select SA's.
12321      * A side effect of setting ixa_ipsec_*_sa is that it will
12322      * be cached in the conn_t.
12323      */
12324     if (ap->ipa_want_esp) {
12325         if (ixa->ixa_ipsec_esp_sa == NULL) {
12326             need_esp_acquire = !ipsec_outbound_sa(mp, ixa,
12327                 IPPROTO_ESP);
12328         }
12329         ASSERT(need_esp_acquire || ixa->ixa_ipsec_esp_sa != NULL);
12330     }

12332     if (ap->ipa_want_ah) {
12333         if (ixa->ixa_ipsec_ah_sa == NULL) {
12334             need_ah_acquire = !ipsec_outbound_sa(mp, ixa,
12335                 IPPROTO_AH);
12336         }
12337         ASSERT(need_ah_acquire || ixa->ixa_ipsec_ah_sa != NULL);

```

```

12338     /*
12339      * The ESP and AH processing order needs to be preserved
12340      * when both protocols are required (ESP should be applied
12341      * before AH for an outbound packet). Force an ESP ACQUIRE
12342      * when both ESP and AH are required, and an AH ACQUIRE
12343      * is needed.
12344      */
12345     if (ap->ipa_want_esp && need_ah_acquire)
12346         need_esp_acquire = B_TRUE;
12347 }

12349 /*
12350  * Send an ACQUIRE (extended, regular, or both) if we need one.
12351  * Release SAs that got referenced, but will not be used until we
12352  * acquire _all_ of the SAs we need.
12353  */
12354     if (need_ah_acquire || need_esp_acquire) {
12355         if (ixa->ixa_ipsec_ah_sa != NULL) {
12356             IPSA_REFRELE(ixa->ixa_ipsec_ah_sa);
12357             ixa->ixa_ipsec_ah_sa = NULL;
12358         }
12359         if (ixa->ixa_ipsec_esp_sa != NULL) {
12360             IPSA_REFRELE(ixa->ixa_ipsec_esp_sa);
12361             ixa->ixa_ipsec_esp_sa = NULL;
12362         }

12364         sadb_acquire(mp, ixa, need_ah_acquire, need_esp_acquire);
12365         return (B_FALSE);
12366     }

12368     return (B_TRUE);
12369 }

12371 /*
12372  * Handle IPsec output processing.
12373  * This function is only entered once for a given packet.
12374  * We try to do things synchronously, but if we need to have user-level
12375  * set up SAs, or ESP or AH uses asynchronous KEF, then the operation
12376  * will be completed
12377  * - when the SAs are added in esp_add_sa_finish/ah_add_sa_finish
12378  * - when asynchronous ESP is done it will do AH
12379  *
12380  * In all cases we come back in ip_output_post_ipsec() to fragment and
12381  * send out the packet.
12382  */
12383 int
12384 ipsec_out_process(mblk_t *mp, ip_xmit_attr_t *ixa)
12385 {
12386     ill_t      *ill = ixa->ixa_nce->nce_ill;
12387     ip_stack_t  *ipst = ixa->ixa_ipst;
12388     ipsec_stack_t *ipss;
12389     ipsec_policy_t *pp;
12390     ipsec_action_t *ap;

12392     ASSERT(ixa->ixa_flags & IXAF_IPSEC_SECURE);

12394     ASSERT((ixa->ixa_ipsec_policy != NULL) ||
12395         (ixa->ixa_ipsec_action != NULL));

12397     ipss = ipst->ips_netstack->netstack_ipsec;
12398     if (!ipsec_loaded(ipss)) {
12399         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12400         ip_drop_packet(mp, B_TRUE, ill,
12401             DROPPER(ipss, ipds_ip_ipsec_not_loaded),
12402             &ipss->ipsec_dropper);
12403         return (ENOTSUP);

```

```

12404     }
12405
12406     ap = ixa->ixa_ipsec_action;
12407     if (ap == NULL) {
12408         pp = ixa->ixa_ipsec_policy;
12409         ASSERT(pp != NULL);
12410         ap = pp->ipsp_act;
12411         ASSERT(ap != NULL);
12412     }
12413
12414     /* Handle explicit drop action and bypass. */
12415     switch (ap->ipa_act.ipa_type) {
12416     case IPSEC_ACT_DISCARD:
12417     case IPSEC_ACT_REJECT:
12418         ip_drop_packet(mp, B_FALSE, ill,
12419             DROPPER(ipss, ipds_spd_explicit), &ipss->ipsec_spd_dropper);
12420         return (EHOSTUNREACH); /* IPsec policy failure */
12421     case IPSEC_ACT_BYPASS:
12422         return (ip_output_post_ipsec(mp, ixa));
12423     }
12424
12425     /*
12426     * The order of processing is first insert a IP header if needed.
12427     * Then insert the ESP header and then the AH header.
12428     */
12429     if ((ixa->ixa_flags & IXAF_IS_IPV4) && ap->ipa_want_se) {
12430         /*
12431         * First get the outer IP header before sending
12432         * it to ESP.
12433         */
12434         ipha_t *oipha, *iipha;
12435         mblk_t *outer_mp, *inner_mp;
12436
12437         if ((outer_mp = allocb(sizeof (ipha_t), BPRI_HI)) == NULL) {
12438             (void) mi_strlog(ill->ill_rq, 0,
12439                 SL_ERROR|SL_TRACE|SL_CONSOLE,
12440                 "ipsec_out_process: "
12441                 "Self-Encapsulation failed: Out of memory\n");
12442             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
12443             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
12444             freemsg(mp);
12445             return (ENOBUFFS);
12446         }
12447         inner_mp = mp;
12448         ASSERT(inner_mp->b_datap->db_type == M_DATA);
12449         oipha = (ipha_t *)outer_mp->b_rptr;
12450         iipha = (ipha_t *)inner_mp->b_rptr;
12451         *oipha = *iipha;
12452         outer_mp->b_wptr += sizeof (ipha_t);
12453         oipha->ipha_length = htons(ntohs(iipha->ipha_length) +
12454             sizeof (ipha_t));
12455         oipha->ipha_protocol = IPPROTO_ENCAP;
12456         oipha->ipha_version_and_hdr_length =
12457             IP_SIMPLE_HDR_VERSION;
12458         oipha->ipha_hdr_checksum = 0;
12459         oipha->ipha_hdr_checksum = ip_csum_hdr(oipha);
12460         outer_mp->b_cont = inner_mp;
12461         mp = outer_mp;
12462
12463         ixa->ixa_flags |= IXAF_IPSEC_TUNNEL;
12464     }
12465
12466     /* If we need to wait for a SA then we can't return any errno */
12467     if (((ap->ipa_want_ah && (ixa->ixa_ipsec_ah_sa == NULL)) ||
12468         (ap->ipa_want_esp && (ixa->ixa_ipsec_esp_sa == NULL))) &&
12469         !ipsec_out_select_sa(mp, ixa))

```

```

12470         return (0);
12471
12472     /*
12473     * By now, we know what SA's to use. Toss over to ESP & AH
12474     * to do the heavy lifting.
12475     */
12476     if (ap->ipa_want_esp) {
12477         ASSERT(ixa->ixa_ipsec_esp_sa != NULL);
12478
12479         mp = ixa->ixa_ipsec_esp_sa->ipsa_output_func(mp, ixa);
12480         if (mp == NULL) {
12481             /*
12482             * Either it failed or is pending. In the former case
12483             * ipIfStatsInDiscards was increased.
12484             */
12485             return (0);
12486         }
12487     }
12488
12489     if (ap->ipa_want_ah) {
12490         ASSERT(ixa->ixa_ipsec_ah_sa != NULL);
12491
12492         mp = ixa->ixa_ipsec_ah_sa->ipsa_output_func(mp, ixa);
12493         if (mp == NULL) {
12494             /*
12495             * Either it failed or is pending. In the former case
12496             * ipIfStatsInDiscards was increased.
12497             */
12498             return (0);
12499         }
12500     }
12501     /*
12502     * We are done with IPsec processing. Send it over
12503     * the wire.
12504     */
12505     return (ip_output_post_ipsec(mp, ixa));
12506 }
12507
12508 /*
12509 * ioctls that go through a down/up sequence may need to wait for the down
12510 * to complete. This involves waiting for the ire and ipif refcnts to go down
12511 * to zero. Subsequently the ioctl is restarted from ipif_ill_refrele_tail.
12512 */
12513 /* ARGSUSED */
12514 void
12515 ip_reprocess_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *dummy_arg)
12516 {
12517     struct iocblk *iocp;
12518     mblk_t *mpl;
12519     ip_ioctl_cmd_t *ipip;
12520     int err;
12521     sin_t *sin;
12522     struct lifreq *lifr;
12523     struct ifreq *ifrr;
12524
12525     iocp = (struct iocblk *)mp->b_rptr;
12526     ASSERT(ipsq != NULL);
12527     /* Existence of mpl verified in ip_wput_nondata */
12528     mpl = mp->b_cont->b_cont;
12529     ipip = ip_ioctl_lookup(iocp->ioc_cmd);
12530     if (ipip->ipi_cmd == SIOCSLIFNAME || ipip->ipi_cmd == IF_UNITSEL) {
12531         /*
12532         * Special case where ipx_current_ipif is not set:
12533         * ill_phyint_reinit merged the v4 and v6 into a single ipsq.
12534         * We are here as were not able to complete the operation in
12535         * ipif_set_values because we could not become exclusive on

```



```

12536         * the new ipsq.
12537         */
12538         ill_t *ill = q->q_ptr;
12539         ipsq_current_start(ipsq, ill->ill_ipif, ipip->ipi_cmd);
12540     }
12541     ASSERT(ipsq->ipsq_xop->ipx_current_ipif != NULL);

12543     if (ipip->ipi_cmd_type == IF_CMD) {
12544         /* This a old style SIOC[GS]IF* command */
12545         ifr = (struct ifreq *)mpl->b_rptr;
12546         sin = (sin_t *)&ifr->ifr_addr;
12547     } else if (ipip->ipi_cmd_type == LIF_CMD) {
12548         /* This a new style SIOC[GS]LIF* command */
12549         lifr = (struct lifreq *)mpl->b_rptr;
12550         sin = (sin_t *)&lifr->lifr_addr;
12551     } else {
12552         sin = NULL;
12553     }

12555     err = (*ipip->ipi_func_restart)(ipsq->ipsq_xop->ipx_current_ipif, sin,
12556         q, mp, ipip, mpl->b_rptr);

12558     DTRACE_PROBE4(ipif__ioctl, char *, "ip_reprocess_ioctl finish",
12559         int, ipip->ipi_cmd,
12560         ill_t *, ipsq->ipsq_xop->ipx_current_ipif->ipif_ill,
12561         ipif_t *, ipsq->ipsq_xop->ipx_current_ipif);

12563     ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);
12564 }

12566 /*
12567  * ioctl processing
12568  *
12569  * ioctl processing starts with ip_ioctl_copyin_setup(), which looks up
12570  * the ioctl command in the ioctl tables, determines the copyin data size
12571  * from the ipi_copyin_size field, and does an mi_copyin() of that size.
12572  *
12573  * ioctl processing then continues when the M_IOCTLDATA makes its way down to
12574  * ip_wput_nondata(). The ioctl is looked up again in the ioctl table, its
12575  * associated 'conn' is refheld till the end of the ioctl and the general
12576  * ioctl processing function ip_process_ioctl() is called to extract the
12577  * arguments and process the ioctl. To simplify extraction, ioctl commands
12578  * are "typed" based on the arguments they take (e.g., LIF_CMD which takes a
12579  * 'struct lifreq'), and a common extract function (e.g., ip_extract_lifreq())
12580  * is used to extract the ioctl's arguments.
12581  *
12582  * ip_process_ioctl determines if the ioctl needs to be serialized, and if
12583  * so goes thru the serialization primitive ipsq_try_enter. Then the
12584  * appropriate function to handle the ioctl is called based on the entry in
12585  * the ioctl table. ioctl completion is encapsulated in ip_ioctl_finish
12586  * which also refreleases the 'conn' that was refheld at the start of the
12587  * ioctl. Finally ipsq_exit is called if needed to exit the ipsq.
12588  *
12589  * Many exclusive ioctls go thru an internal down up sequence as part of
12590  * the operation. For example an attempt to change the IP address of an
12591  * ipif entails ipif_down, set address, ipif_up. Bringing down the interface
12592  * does all the cleanup such as deleting all ires that use this address.
12593  * Then we need to wait till all references to the interface go away.
12594  */
12595 void
12596 ip_process_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *arg)
12597 {
12598     struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
12599     ip_ioctl_cmd_t *ipip = arg;
12600     ip_extract_func_t *extract_funcp;
12601     cmd_info_t ci;

```

```

12602     int err;
12603     boolean_t entered_ipsq = B_FALSE;

12605     ip3dbg(("ip_process_ioctl: ioctl %X\n", iocp->ioc_cmd));

12607     if (ipip == NULL)
12608         ipip = ip_ioctl_lookup(iocp->ioc_cmd);

12610     /*
12611      * SIOCLIFADDIF needs to go thru a special path since the
12612      * ill may not exist yet. This happens in the case of lo0
12613      * which is created using this ioctl.
12614      */
12615     if (ipip->ipi_cmd == SIOCLIFADDIF) {
12616         err = ip_ioctl_addif(NULL, NULL, q, mp, NULL, NULL);
12617         DTRACE_PROBE4(ipif__ioctl, char *, "ip_process_ioctl finish",
12618             int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12619         ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12620         return;
12621     }

12623     ci.ci_ipif = NULL;
12624     switch (ipip->ipi_cmd_type) {
12625     case MISC_CMD:
12626     case MSFILT_CMD:
12627         /*
12628          * All MISC_CMD ioctls come in here -- e.g. SIOCGLIFCONF.
12629          */
12630         if (ipip->ipi_cmd == IF_UNITSEL) {
12631             /* ioctl comes down the ill */
12632             ci.ci_ipif = ((ill_t *)q->q_ptr)->ill_ipif;
12633             ipif_refhold(ci.ci_ipif);
12634         }
12635         err = 0;
12636         ci.ci_sin = NULL;
12637         ci.ci_sin6 = NULL;
12638         ci.ci_lifr = NULL;
12639         extract_funcp = NULL;
12640         break;

12642     case IF_CMD:
12643     case LIF_CMD:
12644         extract_funcp = ip_extract_lifreq;
12645         break;

12647     case ARP_CMD:
12648     case XARP_CMD:
12649         extract_funcp = ip_extract_arpreq;
12650         break;

12652     default:
12653         ASSERT(0);
12654     }

12656     if (extract_funcp != NULL) {
12657         err = (*extract_funcp)(q, mp, ipip, &ci);
12658         if (err != 0) {
12659             DTRACE_PROBE4(ipif__ioctl,
12660                 char *, "ip_process_ioctl finish err",
12661                 int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12662             ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12663             return;
12664         }

12666         /*
12667          * All of the extraction functions return a refheld ipif.

```

```

12668     /*
12669     ASSERT(ci.ci_ipif != NULL);
12670 }

12672 if (!(ipip->ipi_flags & IPI_WR)) {
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,
12679     ci.ci_lifr);
12680     if (ci.ci_ipif != NULL) {
12681         DTRACE_PROBE4(ipif_ioctl,
12682         char *, "ip_process_ioctl finish RD",
12683         int, ipip->ipi_cmd, ill_t *, ci.ci_ipif->ipif_ill,
12684         ipif_t *, ci.ci_ipif);
12685         ipif_refrele(ci.ci_ipif);
12686     } else {
12687         DTRACE_PROBE4(ipif_ioctl,
12688         char *, "ip_process_ioctl finish RD",
12689         int, ipip->ipi_cmd, ill_t *, NULL, ipif_t *, NULL);
12690     }
12691     ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
12692     return;
12693 }

12695 ASSERT(ci.ci_ipif != NULL);

12697 /*
12698 * If ipsq is non-NULL, we are already being called exclusively
12699 */
12700 ASSERT(ipsq == NULL || IAM_WRITER_IPSQ(ipsq));
12701 if (ipsq == NULL) {
12702     ipsq = ipsq_try_enter(ci.ci_ipif, NULL, q, mp, ip_process_ioctl,
12703     NEW_OP, B_TRUE);
12704     if (ipsq == NULL) {
12705         ipif_refrele(ci.ci_ipif);
12706         return;
12707     }
12708     entered_ipsq = B_TRUE;
12709 }
12710 /*
12711 * Release the ipif so that ipif_down and friends that wait for
12712 * references to go away are not misled about the current ipif_refcnt
12713 * values. We are writer so we can access the ipif even after releasing
12714 * the ipif.
12715 */
12716 ipif_refrele(ci.ci_ipif);

12718 ipsq_current_start(ipsq, ci.ci_ipif, ipip->ipi_cmd);

12720 /*
12721 * A return value of EINPROGRESS means the ioctl is
12722 * either queued and waiting for some reason or has
12723 * already completed.
12724 */
12725 err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip, ci.ci_lifr);

12727 DTRACE_PROBE4(ipif_ioctl, char *, "ip_process_ioctl finish WR",
12728 int, ipip->ipi_cmd,
12729 ill_t *, ci.ci_ipif == NULL ? NULL : ci.ci_ipif->ipif_ill,
12730 ipif_t *, ci.ci_ipif);
12731 ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), ipsq);

12733 if (entered_ipsq)

```

```

12734     ipsq_exit(ipsq);
12735 }

12737 /*
12738 * Complete the ioctl. Typically ioctls use the mi package and need to
12739 * do mi_copyout/mi_copy_done.
12740 */
12741 void
12742 ip_ioctl_finish(queue_t *q, mblk_t *mp, int err, int mode, ipsq_t *ipsq)
12743 {
12744     conn_t *connp = NULL;
12745
12746     if (err == EINPROGRESS)
12747         return;
12748
12749     if (CONN_Q(q)) {
12750         connp = Q_TO_CONN(q);
12751         ASSERT(connp->conn_ref >= 2);
12752     }
12753
12754     switch (mode) {
12755     case COPYOUT:
12756         if (err == 0)
12757             mi_copyout(q, mp);
12758         else
12759             mi_copy_done(q, mp, err);
12760         break;
12761
12762     case NO_COPYOUT:
12763         mi_copy_done(q, mp, err);
12764         break;
12765
12766     default:
12767         ASSERT(mode == CONN_CLOSE); /* aborted through CONN_CLOSE */
12768         break;
12769     }
12770
12771     /*
12772     * The conn refhold and ioctlref placed on the conn at the start of the
12773     * ioctl are released here.
12774     */
12775     if (connp != NULL) {
12776         CONN_DEC_IOCTLREF(connp);
12777         CONN_OPER_PENDING_DONE(connp);
12778     }
12779
12780     if (ipsq != NULL)
12781         ipsq_current_finish(ipsq);
12782 }

12784 /* Handles all non data messages */
12785 void
12786 ip_wput_nondata(queue_t *q, mblk_t *mp)
12787 {
12788     mblk_t *mp1;
12789     struct iocblk *iocp;
12790     ip_ioctl_cmd_t *ipip;
12791     conn_t *connp;
12792     cred_t *cr;
12793     char *proto_str;
12794
12795     if (CONN_Q(q))
12796         connp = Q_TO_CONN(q);
12797     else
12798         connp = NULL;

```

```

12800     switch (DB_TYPE(mp)) {
12801     case M_IOCTL:
12802         /*
12803          * IOCTL processing begins in ip_siocctl_copyin_setup which
12804          * will arrange to copy in associated control structures.
12805          */
12806         ip_siocctl_copyin_setup(q, mp);
12807         return;
12808     case M_IOCTLDATA:
12809         /*
12810          * Ensure that this is associated with one of our trans-
12811          * parent ioctls. If it's not ours, discard it if we're
12812          * running as a driver, or pass it on if we're a module.
12813          */
12814         iocp = (struct iocblk *)mp->b_rptr;
12815         ipip = ip_siocctl_lookup(iocp->ioc_cmd);
12816         if (ipip == NULL) {
12817             if (q->q_next == NULL) {
12818                 goto nak;
12819             } else {
12820                 putnext(q, mp);
12821             }
12822             return;
12823         }
12824         if ((q->q_next != NULL) && !(ipip->ipi_flags & IPI_MODOK)) {
12825             /*
12826              * The ioctl is one we recognise, but is not consumed
12827              * by IP as a module and we are a module, so we drop
12828              */
12829             goto nak;
12830         }
12831     }
12832     /* IOCTL continuation following copyin or copyout. */
12833     if (mi_copy_state(q, mp, NULL) == -1) {
12834         /*
12835          * The copy operation failed. mi_copy_state already
12836          * cleaned up, so we're out of here.
12837          */
12838         return;
12839     }
12840     /*
12841      * If we just completed a copy in, we become writer and
12842      * continue processing in ip_siocctl_copyin_done. If it
12843      * was a copy out, we call mi_copyout again. If there is
12844      * nothing more to copy out, it will complete the IOCTL.
12845      */
12846     if (MI_COPY_DIRECTION(mp) == MI_COPY_IN) {
12847         if (!(mpl = mp->b_cont) || !(mpl = mpl->b_cont)) {
12848             mi_copy_done(q, mp, EPROTO);
12849             return;
12850         }
12851         /*
12852          * Check for cases that need more copying. A return
12853          * value of 0 means a second copyin has been started,
12854          * so we return; a return value of 1 means no more
12855          * copying is needed, so we continue.
12856          */
12857         if (ipip->ipi_cmd_type == MSFILT_CMD &&
12858             MI_COPY_COUNT(mp) == 1) {
12859             if (ip_copyin_msfilter(q, mp) == 0)
12860                 return;
12861         }
12862         /*
12863          * Refhold the conn, till the ioctl completes. This is
12864          * needed in case the ioctl ends up in the pending mp
12865          * list. Every mp in the ipx_pending_mp list must have

```

```

12866         * a rehold on the conn to resume processing. The
12867         * rehold is released when the ioctl completes
12868         * (whether normally or abnormally). An iocltref is also
12869         * placed on the conn to prevent TCP from removing the
12870         * queue needed to send the ioctl reply back.
12871         * In all cases ip_ioctl_finish is called to finish
12872         * the ioctl and release the reholds.
12873         */
12874         if (connp != NULL) {
12875             /* This is not a reentry */
12876             CONN_INC_REF(connp);
12877             CONN_INC_IOCTLREF(connp);
12878         } else {
12879             if (!(ipip->ipi_flags & IPI_MODOK)) {
12880                 mi_copy_done(q, mp, EINVAL);
12881                 return;
12882             }
12883         }
12884         ip_process_ioctl(NULL, q, mp, ipip);
12885     } else {
12886         mi_copyout(q, mp);
12887     }
12888     return;
12889 }
12890
12891 case M_IOCNAK:
12892     /*
12893      * The only way we could get here is if a resolver didn't like
12894      * an IOCTL we sent it. This shouldn't happen.
12895      */
12896     (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
12897         "ip_wput_nodata: unexpected M_IOCNAK, ioc_cmd 0x%x",
12898         ((struct iocblk *)mp->b_rptr)->ioc_cmd);
12899     freemsg(mp);
12900     return;
12901 case M_IOCACK:
12902     /* /dev/ip shouldn't see this */
12903     goto nak;
12904 case M_FLUSH:
12905     if (*mp->b_rptr & FLUSHW)
12906         flushq(q, FLUSHALL);
12907     if (q->q_next) {
12908         putnext(q, mp);
12909         return;
12910     }
12911     if (*mp->b_rptr & FLUSHR) {
12912         *mp->b_rptr &= ~FLUSHW;
12913         qreply(q, mp);
12914         return;
12915     }
12916     freemsg(mp);
12917     return;
12918 case M_CTL:
12919     break;
12920 case M_PROTO:
12921 case M_PCPROTO:
12922     /*
12923      * The only PROTO messages we expect are SNMP-related.
12924      */
12925     switch (((union T_primitives *)mp->b_rptr)->type) {
12926     case T_SVR4_OPTMGMT_REQ:
12927         ip2dbg(("ip_wput_nodata: T_SVR4_OPTMGMT_REQ "
12928             "flags 0x%x\n",
12929             ((struct T_optmgmt_req *)mp->b_rptr)->MGMT_flags));
12930     }

```

```

12932         if (connp == NULL) {
12933             proto_str = "T_SVR4_OPTMGMT_REQ";
12934             goto protonak;
12935         }
12936
12937         /*
12938          * All Solaris components should pass a db_credp
12939          * for this TPI message, hence we ASSERT.
12940          * But in case there is some other M_PROTO that looks
12941          * like a TPI message sent by some other kernel
12942          * component, we check and return an error.
12943          */
12944         cr = msg_getcred(mp, NULL);
12945         ASSERT(cr != NULL);
12946         if (cr == NULL) {
12947             mp = mi_tpi_err_ack_alloc(mp, TSYSERR, EINVAL);
12948             if (mp != NULL)
12949                 qreply(q, mp);
12950             return;
12951         }
12952
12953         if (!snmpcom_req(q, mp, ip_snmp_set, ip_snmp_get, cr)) {
12954             proto_str = "Bad SNMPCOM request?";
12955             goto protonak;
12956         }
12957         return;
12958     default:
12959         ipldbg(("ip_wput_nodata: dropping M_PROTO prim %u\n",
12960             (int)*(uint_t *)mp->b_rptr));
12961         freemsg(mp);
12962         return;
12963     }
12964     default:
12965         break;
12966 }
12967 if (q->q_next) {
12968     putnext(q, mp);
12969 } else
12970     freemsg(mp);
12971 return;
12972
12973 nak:
12974     iocp->ioc_error = EINVAL;
12975     mp->b_datap->db_type = M_IOCNAK;
12976     iocp->ioc_count = 0;
12977     qreply(q, mp);
12978     return;
12979
12980 protonak:
12981     cmn_err(CE_NOTE, "IP doesn't process %s as a module", proto_str);
12982     if ((mp = mi_tpi_err_ack_alloc(mp, TPROTO, EINVAL)) != NULL)
12983         qreply(q, mp);
12984 }
12985
12986 /*
12987  * Process IP options in an outbound packet.  Verify that the nexthop in a
12988  * strict source route is onlink.
12989  * Returns non-zero if something fails in which case an ICMP error has been
12990  * sent and mp freed.
12991  *
12992  * Assumes the ULP has called ip_message_options to move nexthop into ipha_dst.
12993  */
12994 int
12995 ip_output_options(mblk_t *mp, ipha_t *ipha, ip_xmit_attr_t *ixa, ill_t *ill)
12996 {
12997     ipoptp_t      opts;

```

```

12998     uchar_t      *opt;
12999     uint8_t      optval;
13000     uint8_t      optlen;
13001     ipaddr_t      dst;
13002     intptr_t      code = 0;
13003     ire_t         *ire;
13004     ip_stack_t     *ipst = ipha->ixa_ipst;
13005     ip_recv_attr_t iras;
13006
13007     ip2dbg(("ip_output_options\n"));
13008
13009     dst = ipha->ipha_dst;
13010     for (optval = ipoptp_first(&opts, ipha);
13011          optval != IPOPT_EOL;
13012          optval = ipoptp_next(&opts)) {
13013         opt = opts.ipoptp_cur;
13014         optlen = opts.ipoptp_len;
13015         ip2dbg(("ip_output_options: opt %d, len %d\n",
13016             optval, optlen));
13017         switch (optval) {
13018             uint32_t off;
13019         case IPOPT_SSRR:
13020         case IPOPT_LSRR:
13021             if ((opts.ipoptp_flags & IPOPT_ERROR) != 0) {
13022                 ipldbg(("ip_output_options: bad option offset\n"));
13023                 code = (char *)&opt[IPOPT_OLEN] -
13024                     (char *)ipha;
13025                 goto param_prob;
13026             }
13027             off = opt[IPOPT_OFFSET];
13028             ipldbg(("ip_output_options: next hop 0x%x\n",
13029                 ntohl(dst)));
13030             /*
13031              * For strict: verify that dst is directly
13032              * reachable.
13033              */
13034             if (optval == IPOPT_SSRR) {
13035                 ire = ire_fhtable_lookup_v4(dst, 0, 0,
13036                     IRE_INTERFACE, NULL, ALL_ZONES,
13037                     ipha->ixa_tsl,
13038                     MATCH_IRE_TYPE | MATCH_IRE_SECATTR, 0, ipst,
13039                     NULL);
13040                 if (ire == NULL) {
13041                     ipldbg(("ip_output_options: SSRR not"
13042                         " directly reachable: 0x%x\n",
13043                         ntohl(dst)));
13044                     goto bad_src_route;
13045                 }
13046                 ire_refrele(ire);
13047             }
13048             break;
13049         case IPOPT_RR:
13050             if ((opts.ipoptp_flags & IPOPT_ERROR) != 0) {
13051                 ipldbg(("ip_output_options: bad option offset\n"));
13052                 code = (char *)&opt[IPOPT_OLEN] -
13053                     (char *)ipha;
13054                 goto param_prob;
13055             }
13056             break;
13057         case IPOPT_TS:
13058             /*
13059              * Verify that length >= 5 and that there is either
13060              * room for another timestamp or that the overflow
13061              * counter is not maxed out.

```

```

13064     /*
13065     code = (char *)&opt[IPOPT_OLEN] - (char *)ipha;
13066     if (optlen < IPOPT_MINLEN_IT) {
13067         goto param_prob;
13068     }
13069     if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
13070         ipldbg(("ip_output_options: bad option offset\n"));
13071         code = (char *)&opt[IPOPT_OFFSET] -
13072             (char *)ipha;
13073         goto param_prob;
13074     }
13075     switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
13076     case IPOPT_TS_TSONLY:
13077         off = IPOPT_TS_TIMELEN;
13078         break;
13079     case IPOPT_TS_TSANDADDR:
13080     case IPOPT_TS_PRESPEC:
13081     case IPOPT_TS_PRESPEC RFC791:
13082         off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
13083         break;
13084     default:
13085         code = (char *)&opt[IPOPT_POS_OV_FLG] -
13086             (char *)ipha;
13087         goto param_prob;
13088     }
13089     if (opt[IPOPT_OFFSET] - 1 + off > optlen &&
13090         (opt[IPOPT_POS_OV_FLG] & 0xF0) == 0xF0) {
13091         /*
13092          * No room and the overflow counter is 15
13093          * already.
13094          */
13095         goto param_prob;
13096     }
13097     break;
13098 }
13099 }
13100 }

13102 if ((opts.ipoptp_flags & IPOPTP_ERROR) == 0)
13103     return (0);

13105 ipldbg(("ip_output_options: error processing IP options."));
13106 code = (char *)&opt[IPOPT_OFFSET] - (char *)ipha;

13108 param_prob:
13109     bzero(&iras, sizeof (iras));
13110     iras.ira_ill = iras.ira_rill = ill;
13111     iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13112     iras.ira_rifindex = iras.ira_ruifindex;
13113     iras.ira_flags = IRAF_IS_IPV4;

13115 ip_drop_output("ip_output_options", mp, ill);
13116 icmp_param_problem(mp, (uint8_t)code, &iras);
13117 ASSERT(! (iras.ira_flags & IRAF_IPSEC_SECURE));
13118 return (-1);

13120 bad_src_route:
13121     bzero(&iras, sizeof (iras));
13122     iras.ira_ill = iras.ira_rill = ill;
13123     iras.ira_ruifindex = ill->ill_phyint->phyint_ifindex;
13124     iras.ira_rifindex = iras.ira_ruifindex;
13125     iras.ira_flags = IRAF_IS_IPV4;

13127 ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
13128 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, &iras);
13129 ASSERT(! (iras.ira_flags & IRAF_IPSEC_SECURE));

```

```

13130     return (-1);
13131 }

13133 /*
13134  * The maximum value of conn_drain_list_cnt is CONN_MAXDRAINCNT.
13135  * conn_drain_list_cnt can be changed by setting conn_drain_nthreads
13136  * thru /etc/system.
13137  */
13138 #define CONN_MAXDRAINCNT        64

13140 static void
13141 conn_drain_init(ip_stack_t *ipst)
13142 {
13143     int i, j;
13144     idl_tx_list_t *itl_tx;

13146     ipst->ips_conn_drain_list_cnt = conn_drain_nthreads;

13148     if ((ipst->ips_conn_drain_list_cnt == 0) ||
13149         (ipst->ips_conn_drain_list_cnt > CONN_MAXDRAINCNT)) {
13150         /*
13151          * Default value of the number of drainers is the
13152          * number of cpus, subject to maximum of 8 drainers.
13153          */
13154         if (boot_max_ncpus != -1)
13155             ipst->ips_conn_drain_list_cnt = MIN(boot_max_ncpus, 8);
13156         else
13157             ipst->ips_conn_drain_list_cnt = MIN(max_ncpus, 8);
13158     }

13160     ipst->ips_idl_tx_list =
13161         kmem_zalloc(TX_FANOUT_SIZE * sizeof (idl_tx_list_t), KM_SLEEP);
13162     for (i = 0; i < TX_FANOUT_SIZE; i++) {
13163         itl_tx = &ipst->ips_idl_tx_list[i];
13164         itl_tx->txl_drain_list =
13165             kmem_zalloc(ipst->ips_conn_drain_list_cnt *
13166                 sizeof (idl_t), KM_SLEEP);
13167         mutex_init(&itl_tx->txl_lock, NULL, MUTEX_DEFAULT, NULL);
13168         for (j = 0; j < ipst->ips_conn_drain_list_cnt; j++) {
13169             mutex_init(&itl_tx->txl_drain_list[j].idl_lock, NULL,
13170                 MUTEX_DEFAULT, NULL);
13171             itl_tx->txl_drain_list[j].idl_itl = itl_tx;
13172         }
13173     }
13174 }

13176 static void
13177 conn_drain_fini(ip_stack_t *ipst)
13178 {
13179     int i;
13180     idl_tx_list_t *itl_tx;

13182     for (i = 0; i < TX_FANOUT_SIZE; i++) {
13183         itl_tx = &ipst->ips_idl_tx_list[i];
13184         kmem_free(itl_tx->txl_drain_list,
13185             ipst->ips_conn_drain_list_cnt * sizeof (idl_t));
13186     }
13187     kmem_free(ipst->ips_idl_tx_list,
13188         TX_FANOUT_SIZE * sizeof (idl_tx_list_t));
13189     ipst->ips_idl_tx_list = NULL;
13190 }

13192 /*
13193  * Flow control has blocked us from proceeding. Insert the given conn in one
13194  * of the conn drain lists. When flow control is unblocked, either ip_wsrv()
13195  * (STREAMS) or ill_flow_enable() (direct) will be called back, which in turn

```

```

13196 * will call conn_walk_drain(). See the flow control notes at the top of this
13197 * file for more details.
13198 */
13199 void
13200 conn_drain_insert(conn_t *connp, idl_tx_list_t *tx_list)
13201 {
13202     idl_t *idl = tx_list->txl_drain_list;
13203     uint_t index;
13204     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;

13206     mutex_enter(&connp->conn_lock);
13207     if (connp->conn_state_flags & CONN_CLOSING) {
13208         /*
13209          * The conn is closing as a result of which CONN_CLOSING
13210          * is set. Return.
13211          */
13212         mutex_exit(&connp->conn_lock);
13213         return;
13214     } else if (connp->conn_idl == NULL) {
13215         /*
13216          * Assign the next drain list round robin. We don't use
13217          * a lock, and thus it may not be strictly round robin.
13218          * Atomicity of load/stores is enough to make sure that
13219          * conn_drain_list_index is always within bounds.
13220          */
13221         index = tx_list->txl_drain_index;
13222         ASSERT(index < ipst->ips_conn_drain_list_cnt);
13223         connp->conn_idl = &tx_list->txl_drain_list[index];
13224         index++;
13225         if (index == ipst->ips_conn_drain_list_cnt)
13226             index = 0;
13227         tx_list->txl_drain_index = index;
13228     } else {
13229         ASSERT(connp->conn_idl->idl_itl == tx_list);
13230     }
13231     mutex_exit(&connp->conn_lock);

13233     idl = connp->conn_idl;
13234     mutex_enter(&idl->idl_lock);
13235     if ((connp->conn_drain_prev != NULL) ||
13236         (connp->conn_state_flags & CONN_CLOSING)) {
13237         /*
13238          * The conn is either already in the drain list or closing.
13239          * (We needed to check for CONN_CLOSING again since close can
13240          * sneak in between dropping conn_lock and acquiring idl_lock.)
13241          */
13242         mutex_exit(&idl->idl_lock);
13243         return;
13244     }

13246     /*
13247     * The conn is not in the drain list. Insert it at the
13248     * tail of the drain list. The drain list is circular
13249     * and doubly linked. idl_conn points to the 1st element
13250     * in the list.
13251     */
13252     if (idl->idl_conn == NULL) {
13253         idl->idl_conn = connp;
13254         connp->conn_drain_next = connp;
13255         connp->conn_drain_prev = connp;
13256     } else {
13257         conn_t *head = idl->idl_conn;

13259         connp->conn_drain_next = head;
13260         connp->conn_drain_prev = head->conn_drain_prev;
13261         head->conn_drain_prev->conn_drain_next = connp;

```

```

13262         head->conn_drain_prev = connp;
13263     }
13264     /*
13265     * For non streams based sockets assert flow control.
13266     */
13267     conn_setqfull(connp, NULL);
13268     mutex_exit(&idl->idl_lock);
13269 }

13271 static void
13272 conn_drain_remove(conn_t *connp)
13273 {
13274     idl_t *idl = connp->conn_idl;

13276     if (idl != NULL) {
13277         /*
13278          * Remove ourselves from the drain list.
13279          */
13280         if (connp->conn_drain_next == connp) {
13281             /* Singleton in the list */
13282             ASSERT(connp->conn_drain_prev == connp);
13283             idl->idl_conn = NULL;
13284         } else {
13285             connp->conn_drain_prev->conn_drain_next =
13286                 connp->conn_drain_next;
13287             connp->conn_drain_next->conn_drain_prev =
13288                 connp->conn_drain_prev;
13289             if (idl->idl_conn == connp)
13290                 idl->idl_conn = connp->conn_drain_next;
13291         }

13293         /*
13294          * NOTE: because conn_idl is associated with a specific drain
13295          * list which in turn is tied to the index the TX ring
13296          * (txl_cookie) hashes to, and because the TX ring can change
13297          * over the lifetime of the conn_t, we must clear conn_idl so
13298          * a subsequent conn_drain_insert() will set conn_idl again
13299          * based on the latest txl_cookie.
13300          */
13301         connp->conn_idl = NULL;
13302     }
13303     connp->conn_drain_next = NULL;
13304     connp->conn_drain_prev = NULL;

13306     conn_clrqfull(connp, NULL);
13307     /*
13308     * For streams based sockets open up flow control.
13309     */
13310     if (!IPCL_IS_NONSTR(connp))
13311         enableleak(connp->conn_wq);
13312 }

13314 /*
13315 * This conn is closing, and we are called from ip_close. OR
13316 * this conn is draining because flow-control on the ill has been relieved.
13317 */
13318 * We must also need to remove conn's on this idl from the list, and also
13319 * inform the sockfs upcalls about the change in flow-control.
13320 */
13321 static void
13322 conn_drain(conn_t *connp, boolean_t closing)
13323 {
13324     idl_t *idl;
13325     conn_t *next_connp;

13327     /*

```

```

13328     * connp->conn_idl is stable at this point, and no lock is needed
13329     * to check it. If we are called from ip_close, close has already
13330     * set CONN_CLOSING, thus freezing the value of conn_idl, and
13331     * called us only because conn_idl is non-null. If we are called thru
13332     * service, conn_idl could be null, but it cannot change because
13333     * service is single-threaded per queue, and there cannot be another
13334     * instance of service trying to call conn_drain_insert on this conn
13335     * now.
13336     */
13337     ASSERT(!closing || connp == NULL || connp->conn_idl != NULL);

13339     /*
13340     * If the conn doesn't exist or is not on a drain list, bail.
13341     */
13342     if (connp == NULL || connp->conn_idl == NULL ||
13343         connp->conn_drain_prev == NULL) {
13344         return;
13345     }

13347     idl = connp->conn_idl;
13348     ASSERT(MUTEX_HELD(&idl->idl_lock));

13350     if (!closing) {
13351         next_connp = connp->conn_drain_next;
13352         while (next_connp != connp) {
13353             conn_t *delconnp = next_connp;

13355             next_connp = next_connp->conn_drain_next;
13356             conn_drain_remove(delconnp);
13357         }
13358         ASSERT(connp->conn_drain_next == idl->idl_conn);
13359     }
13360     conn_drain_remove(connp);
13361 }

13363 /*
13364 * Write service routine. Shared perimeter entry point.
13365 * The device queue's messages has fallen below the low water mark and STREAMS
13366 * has backenabled the ill_wq. Send sockfs notification about flow-control on
13367 * each waiting conn.
13368 */
13369 void
13370 ip_wsrv(queue_t *q)
13371 {
13372     ill_t *ill;

13374     ill = (ill_t *)q->q_ptr;
13375     if (ill->ill_state_flags == 0) {
13376         ip_stack_t *ipst = ill->ill_ipst;

13378         /*
13379         * The device flow control has opened up.
13380         * Walk through conn drain lists and enable the
13381         * first conn in each list. This makes sense only
13382         * if the stream is fully plumbed and setup.
13383         * Hence the ill_state_flags check above.
13384         */
13385         ipldbg(("ip_wsrv: walking\n"));
13386         conn_walk_drain(ipst, &ipst->ips_idl_tx_list[0]);
13387         enableok(ill->ill_wq);
13388     }
13389 }

13391 /*
13392 * Callback to disable flow control in IP.
13393 */

```

```

13394 * This is a mac client callback added when the DLD_CAPAB_DIRECT capability
13395 * is enabled.
13396 */
13397 * When MAC_TX() is not able to send any more packets, dld sets its queue
13398 * to QFULL and enable the STREAMS flow control. Later, when the underlying
13399 * driver is able to continue to send packets, it calls mac_tx_(ring_)update()
13400 * function and wakes up corresponding mac worker threads, which in turn
13401 * calls this callback function, and disables flow control.
13402 */
13403 void
13404 ill_flow_enable(void *arg, ip_mac_tx_cookie_t cookie)
13405 {
13406     ill_t *ill = (ill_t *)arg;
13407     ip_stack_t *ipst = ill->ill_ipst;
13408     idl_tx_list_t *idl_txl;

13410     idl_txl = &ipst->ips_idl_tx_list[IDLHASHINDEX(cookie)];
13411     mutex_enter(&idl_txl->txl_lock);
13412     /* add code to to set a flag to indicate idl_txl is enabled */
13413     conn_walk_drain(ipst, idl_txl);
13414     mutex_exit(&idl_txl->txl_lock);
13415 }

13417 /*
13418 * Flow control has been relieved and STREAMS has backenabled us; drain
13419 * all the conn lists on 'tx_list'.
13420 */
13421 static void
13422 conn_walk_drain(ip_stack_t *ipst, idl_tx_list_t *tx_list)
13423 {
13424     int i;
13425     idl_t *idl;

13427     IP_STAT(ipst, ip_conn_walk_drain);

13429     for (i = 0; i < ipst->ips_conn_drain_list_cnt; i++) {
13430         idl = &tx_list->txl_drain_list[i];
13431         mutex_enter(&idl->idl_lock);
13432         conn_drain(idl->idl_conn, B_FALSE);
13433         mutex_exit(&idl->idl_lock);
13434     }
13435 }

13437 /*
13438 * Determine if the ill and multicast aspects of that packets
13439 * "matches" the conn.
13440 */
13441 boolean_t
13442 conn_wantpacket(conn_t *connp, ip_recv_attr_t *ira, ipha_t *ipha)
13443 {
13444     ill_t *ill = ira->ira_rill;
13445     zoneid_t zoneid = ira->ira_zoneid;
13446     uint_t in_ifindex;
13447     ipaddr_t dst, src;

13449     dst = ipha->ipha_dst;
13450     src = ipha->ipha_src;

13452     /*
13453     * conn_incoming_ifindex is set by IP_BOUND_IF which limits
13454     * unicast, broadcast and multicast reception to
13455     * conn_incoming_ifindex.
13456     * conn_wantpacket is called for unicast, broadcast and
13457     * multicast packets.
13458     */
13459     in_ifindex = connp->conn_incoming_ifindex;

```

```

13461      /* mpathd can bind to the under IPMP interface, which we allow */
13462      if (in_ifindex != 0 && in_ifindex != ill->ill_phyint->phyint_ifindex) {
13463          if (!IS_UNDER_IPMP(ill))
13464              return (B_FALSE);
13466          if (in_ifindex != ipmp_ill_get_ipmp_ifindex(ill))
13467              return (B_FALSE);
13468      }
13470      if (!IPCL_ZONE_MATCH(connp, zoneid))
13471          return (B_FALSE);
13473      if (!(ira->ira_flags & IRAF_MULTICAST))
13474          return (B_TRUE);
13476      if (connp->conn_multi_router) {
13477          /* multicast packet and multicast router socket: send up */
13478          return (B_TRUE);
13479      }
13481      if (ipha->ipha_protocol == IPPROTO_PIM ||
13482          ipha->ipha_protocol == IPPROTO_RSVP)
13483          return (B_TRUE);
13485      return (conn_hasmembers_ill_withsrc_v4(connp, dst, src, ira->ira_ill));
13486 }
13488 void
13489 conn_setqfull(conn_t *connp, boolean_t *flow_stopped)
13490 {
13491     if (IPCL_IS_NONSTR(connp)) {
13492         (*connp->conn_upcalls->su_txq_full)
13493             (connp->conn_upper_handle, B_TRUE);
13494         if (flow_stopped != NULL)
13495             *flow_stopped = B_TRUE;
13496     } else {
13497         queue_t *q = connp->conn_wq;
13499         ASSERT(q != NULL);
13500         if (!(q->q_flag & QFULL)) {
13501             mutex_enter(QLOCK(q));
13502             if (!(q->q_flag & QFULL)) {
13503                 /* still need to set QFULL */
13504                 q->q_flag |= QFULL;
13505                 /* set flow_stopped to true under QLOCK */
13506                 if (flow_stopped != NULL)
13507                     *flow_stopped = B_TRUE;
13508                 mutex_exit(QLOCK(q));
13509             } else {
13510                 /* flow_stopped is left unchanged */
13511                 mutex_exit(QLOCK(q));
13512             }
13513         }
13514     }
13515 }
13517 void
13518 conn_clrqfull(conn_t *connp, boolean_t *flow_stopped)
13519 {
13520     if (IPCL_IS_NONSTR(connp)) {
13521         (*connp->conn_upcalls->su_txq_full)
13522             (connp->conn_upper_handle, B_FALSE);
13523         if (flow_stopped != NULL)
13524             *flow_stopped = B_FALSE;
13525     } else {

```

```

13526         queue_t *q = connp->conn_wq;
13528         ASSERT(q != NULL);
13529         if (q->q_flag & QFULL) {
13530             mutex_enter(QLOCK(q));
13531             if (q->q_flag & QFULL) {
13532                 q->q_flag &= ~QFULL;
13533                 /* set flow_stopped to false under QLOCK */
13534                 if (flow_stopped != NULL)
13535                     *flow_stopped = B_FALSE;
13536                 mutex_exit(QLOCK(q));
13537                 if (q->q_flag & QWANTW)
13538                     qbackenable(q, 0);
13539             } else {
13540                 /* flow_stopped is left unchanged */
13541                 mutex_exit(QLOCK(q));
13542             }
13543         }
13544     }
13546     mutex_enter(&connp->conn_lock);
13547     connp->conn_blocked = B_FALSE;
13548     mutex_exit(&connp->conn_lock);
13549 }
13551 /*
13552  * Return the length in bytes of the IPv4 headers (base header, label, and
13553  * other IP options) that will be needed based on the
13554  * ip_pkt_t structure passed by the caller.
13555  *
13556  * The returned length does not include the length of the upper level
13557  * protocol (ULP) header.
13558  * The caller needs to check that the length doesn't exceed the max for IPv4.
13559  */
13560 int
13561 ip_total_hdrs_len_v4(const ip_pkt_t *ipp)
13562 {
13563     int len;
13565     len = IP_SIMPLE_HDR_LENGTH;
13566     if (ipp->ipp_fields & IPPF_LABEL_V4) {
13567         ASSERT(ipp->ipp_label_len_v4 != 0);
13568         /* We need to round up here */
13569         len += (ipp->ipp_label_len_v4 + 3) & ~3;
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         len += ipp->ipp_ipv4_options_len;
13576     }
13577     return (len);
13578 }
13580 /*
13581  * All-purpose routine to build an IPv4 header with options based
13582  * on the abstract ip_pkt_t.
13583  *
13584  * The caller has to set the source and destination address as well as
13585  * ipha_length. The caller has to massage any source route and compensate
13586  * for the ULP pseudo-header checksum due to the source route.
13587  */
13588 void
13589 ip_build_hdrs_v4(uchar_t *buf, uint_t buf_len, const ip_pkt_t *ipp,
13590     uint8_t protocol)
13591 {

```



```

13592     ipha_t *ipha = (ipha_t *)buf;
13593     uint8_t *cp;

13595     /* Initialize IPv4 header */
13596     ipha->ipha_type_of_service = ipp->ipp_type_of_service;
13597     ipha->ipha_length = 0; /* Caller will set later */
13598     ipha->ipha_ident = 0;
13599     ipha->ipha_fragment_offset_and_flags = 0;
13600     ipha->ipha_ttl = ipp->ipp_unicast_hops;
13601     ipha->ipha_protocol = protocol;
13602     ipha->ipha_hdr_checksum = 0;

13604     if ((ipp->ipp_fields & IPPF_ADDR) &&
13605         IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
13606         ipha->ipha_src = ipp->ipp_addr_v4;

13608     cp = (uint8_t *)&ipha[1];
13609     if (ipp->ipp_fields & IPPF_LABEL_V4) {
13610         ASSERT(ipp->ipp_label_len_v4 != 0);
13611         bcopy(ipp->ipp_label_v4, cp, ipp->ipp_label_len_v4);
13612         cp += ipp->ipp_label_len_v4;
13613         /* We need to round up here */
13614         while ((uintptr_t)cp & 0x3) {
13615             *cp++ = IPOPT_NOP;
13616         }
13617     }

13619     if (ipp->ipp_fields & IPPF_IPV4_OPTIONS) {
13620         ASSERT(ipp->ipp_ipv4_options_len != 0);
13621         ASSERT((ipp->ipp_ipv4_options_len & 3) == 0);
13622         bcopy(ipp->ipp_ipv4_options, cp, ipp->ipp_ipv4_options_len);
13623         cp += ipp->ipp_ipv4_options_len;
13624     }
13625     ipha->ipha_version_and_hdr_length =
13626         (uint8_t)((IP_VERSION << 4) + buf_len / 4);

13628     ASSERT((int)(cp - buf) == buf_len);
13629 }

13631 /* Allocate the private structure */
13632 static int
13633 ip_priv_alloc(void **bufp)
13634 {
13635     void *buf;

13637     if ((buf = kmem_alloc(sizeof (ip_priv_t), KM_NOSLEEP)) == NULL)
13638         return (ENOMEM);

13640     *bufp = buf;
13641     return (0);
13642 }

13644 /* Function to delete the private structure */
13645 void
13646 ip_priv_free(void *buf)
13647 {
13648     ASSERT(buf != NULL);
13649     kmem_free(buf, sizeof (ip_priv_t));
13650 }

13652 /*
13653  * The entry point for IPPF processing.
13654  * If the classifier (IPGPC_CLASSIFY) is not loaded and configured, the
13655  * routine just returns.
13656  *
13657  * When called, ip_process generates an ipp_packet_t structure

```

```

13658  * which holds the state information for this packet and invokes the
13659  * the classifier (via ipp_packet_process). The classification, depending on
13660  * configured filters, results in a list of actions for this packet. Invoking
13661  * an action may cause the packet to be dropped, in which case we return NULL.
13662  * proc indicates the callout position for
13663  * this packet and ill is the interface this packet arrived on or will leave
13664  * on (inbound and outbound resp.).
13665  *
13666  * We do the processing on the rill (mapped to the upper if ipmp), but MIB
13667  * on the ill corresponding to the destination IP address.
13668  */
13669 mblk_t *
13670 ip_process(ip_proc_t proc, mblk_t *mp, ill_t *rill, ill_t *ill)
13671 {
13672     ip_priv_t *priv;
13673     ipp_action_id_t aid;
13674     int rc = 0;
13675     ipp_packet_t *pp;

13677     /* If the classifier is not loaded, return */
13678     if ((aid = ipp_action_lookup(IPGPC_CLASSIFY)) == IPP_ACTION_INVALID) {
13679         return (mp);
13680     }

13682     ASSERT(mp != NULL);

13684     /* Allocate the packet structure */
13685     rc = ipp_packet_alloc(&pp, "ip", aid);
13686     if (rc != 0)
13687         goto drop;

13689     /* Allocate the private structure */
13690     rc = ip_priv_alloc((void **)&priv);
13691     if (rc != 0) {
13692         ipp_packet_free(pp);
13693         goto drop;
13694     }
13695     priv->proc = proc;
13696     priv->ill_index = ill_get_upper_ifindex(rill);

13698     ipp_packet_set_private(pp, priv, ip_priv_free);
13699     ipp_packet_set_data(pp, mp);

13701     /* Invoke the classifier */
13702     rc = ipp_packet_process(&pp);
13703     if (pp != NULL) {
13704         mp = ipp_packet_get_data(pp);
13705         ipp_packet_free(pp);
13706         if (rc != 0)
13707             goto drop;
13708         return (mp);
13709     } else {
13710         /* No mp to trace in ip_drop_input/ip_drop_output */
13711         mp = NULL;
13712     }

13713 drop:
13714     if (proc == IPP_LOCAL_IN || proc == IPP_FWD_IN) {
13715         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
13716         ip_drop_input("ip_process", mp, ill);
13717     } else {
13718         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
13719         ip_drop_output("ip_process", mp, ill);
13720     }
13721     freemsg(mp);
13722     return (NULL);
13723 }

```

```

13725 /*
13726  * Propagate a multicast group membership operation (add/drop) on
13727  * all the interfaces crossed by the related multirt routes.
13728  * The call is considered successful if the operation succeeds
13729  * on at least one interface.
13730  */
13731  * This assumes that a set of IRE_HOST/RTF_MULTIRT has been created for the
13732  * multicast addresses with the ire argument being the first one.
13733  * We walk the bucket to find all the of those.
13734  *
13735  * Common to IPv4 and IPv6.
13736  */
13737 static int
13738 ip_multirt_apply_membership(int (*fn)(conn_t *, boolean_t,
13739   const in6_addr_t *, ipaddr_t, uint_t, mcast_record_t, const in6_addr_t *),
13740   ire_t *ire, conn_t *connp, boolean_t checkonly, const in6_addr_t *v6group,
13741   mcast_record_t fmode, const in6_addr_t *v6src)
13742 {
13743     ire_t      *ire_gw;
13744     irb_t      *irb;
13745     int         ifindex;
13746     int         error = 0;
13747     int         result;
13748     ip_stack_t *ipst = ire->ire_ipst;
13749     ipaddr_t    group;
13750     boolean_t   isv6;
13751     int         match_flags;

13753     if (IN6_IS_ADDR_V4MAPPED(v6group)) {
13754         IN6_V4MAPPED_TO_IPADDR(v6group, group);
13755         isv6 = B_FALSE;
13756     } else {
13757         isv6 = B_TRUE;
13758     }

13760     irb = ire->ire_bucket;
13761     ASSERT(irb != NULL);

13763     result = 0;
13764     irb_refhold(irb);
13765     for (; ire != NULL; ire = ire->ire_next) {
13766         if ((ire->ire_flags & RTF_MULTIRT) == 0)
13767             continue;

13769         /* We handle -ifp routes by matching on the ill if set */
13770         match_flags = MATCH_IRE_TYPE;
13771         if (ire->ire_ill != NULL)
13772             match_flags |= MATCH_IRE_ILL;

13774         if (isv6) {
13775             if (!IN6_ARE_ADDR_EQUAL(&ire->ire_addr_v6, v6group))
13776                 continue;

13778             ire_gw = ire_ftable_lookup_v6(&ire->ire_gateway_addr_v6,
13779                 0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13780                 match_flags, 0, ipst, NULL);
13781         } else {
13782             if (ire->ire_addr != group)
13783                 continue;

13785             ire_gw = ire_ftable_lookup_v4(ire->ire_gateway_addr,
13786                 0, 0, IRE_INTERFACE, ire->ire_ill, ALL_ZONES, NULL,
13787                 match_flags, 0, ipst, NULL);
13788         }
13789         /* No interface route exists for the gateway; skip this ire. */

```

```

13790         if (ire_gw == NULL)
13791             continue;
13792         if (ire_gw->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
13793             ire_refrele(ire_gw);
13794             continue;
13795         }
13796         ASSERT(ire_gw->ire_ill != NULL); /* IRE_INTERFACE */
13797         ifindex = ire_gw->ire_ill->ill_phyint->phyint_ifindex;

13799         /*
13800          * The operation is considered a success if
13801          * it succeeds at least once on any one interface.
13802          */
13803         error = fn(connp, checkonly, v6group, INADDR_ANY, ifindex,
13804             fmode, v6src);
13805         if (error == 0)
13806             result = CGTP_MCAST_SUCCESS;

13808         ire_refrele(ire_gw);
13809     }
13810     irb_refrele(irb);
13811     /*
13812      * Consider the call as successful if we succeeded on at least
13813      * one interface. Otherwise, return the last encountered error.
13814      */
13815     return (result == CGTP_MCAST_SUCCESS ? 0 : error);
13816 }

13818 /*
13819  * Return the expected CGTP hooks version number.
13820  */
13821 int
13822 ip_cgtp_filter_supported(void)
13823 {
13824     return (ip_cgtp_filter_rev);
13825 }

13827 /*
13828  * CGTP hooks can be registered by invoking this function.
13829  * Checks that the version number matches.
13830  */
13831 int
13832 ip_cgtp_filter_register(netstackid_t stackid, cgtp_filter_ops_t *ops)
13833 {
13834     netstack_t *ns;
13835     ip_stack_t *ipst;

13837     if (ops->cfo_filter_rev != CGTP_FILTER_REV)
13838         return (ENOTSUP);

13840     ns = netstack_find_by_stackid(stackid);
13841     if (ns == NULL)
13842         return (EINVAL);
13843     ipst = ns->netstack_ip;
13844     ASSERT(ipst != NULL);

13846     if (ipst->ips_ip_cgtp_filter_ops != NULL) {
13847         netstack_rele(ns);
13848         return (EALREADY);
13849     }

13851     ipst->ips_ip_cgtp_filter_ops = ops;

13853     ill_set_inputfn_all(ipst);

13855     netstack_rele(ns);

```

```

13856     return (0);
13857 }

13859 /*
13860  * CGTP hooks can be unregistered by invoking this function.
13861  * Returns ENXIO if there was no registration.
13862  * Returns EBUSY if the ndd variable has not been turned off.
13863  */
13864 int
13865 ip_cgtp_filter_unregister(netstackid_t stackid)
13866 {
13867     netstack_t *ns;
13868     ip_stack_t *ipst;

13870     ns = netstack_find_by_stackid(stackid);
13871     if (ns == NULL)
13872         return (EINVAL);
13873     ipst = ns->netstack_ip;
13874     ASSERT(ipst != NULL);

13876     if (ipst->ips_ip_cgtp_filter) {
13877         netstack_rele(ns);
13878         return (EBUSY);
13879     }

13881     if (ipst->ips_ip_cgtp_filter_ops == NULL) {
13882         netstack_rele(ns);
13883         return (ENXIO);
13884     }
13885     ipst->ips_ip_cgtp_filter_ops = NULL;

13887     ill_set_inputfn_all(ipst);

13889     netstack_rele(ns);
13890     return (0);
13891 }

13893 /*
13894  * Check whether there is a CGTP filter registration.
13895  * Returns non-zero if there is a registration, otherwise returns zero.
13896  * Note: returns zero if bad stackid.
13897  */
13898 int
13899 ip_cgtp_filter_is_registered(netstackid_t stackid)
13900 {
13901     netstack_t *ns;
13902     ip_stack_t *ipst;
13903     int ret;

13905     ns = netstack_find_by_stackid(stackid);
13906     if (ns == NULL)
13907         return (0);
13908     ipst = ns->netstack_ip;
13909     ASSERT(ipst != NULL);

13911     if (ipst->ips_ip_cgtp_filter_ops != NULL)
13912         ret = 1;
13913     else
13914         ret = 0;

13916     netstack_rele(ns);
13917     return (ret);
13918 }

13920 static int
13921 ip_squeue_switch(int val)

```

```

13922 {
13923     int rval;

13925     switch (val) {
13926     case IP_SQUEUE_ENTER_NODRAIN:
13927         rval = SQ_NODRAIN;
13928         break;
13929     case IP_SQUEUE_ENTER:
13930         rval = SQ_PROCESS;
13931         break;
13932     case IP_SQUEUE_FILL:
13933     default:
13934         rval = SQ_FILL;
13935         break;
13936     }
13937     return (rval);
13938 }

13940 static void *
13941 ip_kstat2_init(netstackid_t stackid, ip_stat_t *ip_statisticsp)
13942 {
13943     kstat_t *ksp;

13945     ip_stat_t template = {
13946         "ip_udp_fannorm",          KSTAT_DATA_UINT64 },
13947         "ip_udp_fanmb",           KSTAT_DATA_UINT64 },
13948         "ip_rcv_pullup",          KSTAT_DATA_UINT64 },
13949         "ip_db_ref",              KSTAT_DATA_UINT64 },
13950         "ip_notaligned",          KSTAT_DATA_UINT64 },
13951         "ip_multimblk",           KSTAT_DATA_UINT64 },
13952         "ip_opt",                 KSTAT_DATA_UINT64 },
13953         "ipsec_proto_ahesp",      KSTAT_DATA_UINT64 },
13954         "ip_conn_flputbq",        KSTAT_DATA_UINT64 },
13955         "ip_conn_walk_drain",     KSTAT_DATA_UINT64 },
13956         "ip_out_sw_cksum",        KSTAT_DATA_UINT64 },
13957         "ip_out_sw_cksum_bytes",  KSTAT_DATA_UINT64 },
13958         "ip_in_sw_cksum",         KSTAT_DATA_UINT64 },
13959         "ip_ire_reclaim_calls",   KSTAT_DATA_UINT64 },
13960         "ip_ire_reclaim_deleted", KSTAT_DATA_UINT64 },
13961         "ip_nce_reclaim_calls",   KSTAT_DATA_UINT64 },
13962         "ip_nce_reclaim_deleted", KSTAT_DATA_UINT64 },
13963         "ip_dce_reclaim_calls",   KSTAT_DATA_UINT64 },
13964         "ip_dce_reclaim_deleted", KSTAT_DATA_UINT64 },
13965         "ip_tcp_in_full_hw_cksum_err", KSTAT_DATA_UINT64 },
13966         "ip_tcp_in_part_hw_cksum_err", KSTAT_DATA_UINT64 },
13967         "ip_tcp_in_sw_cksum_err",  KSTAT_DATA_UINT64 },
13968         "ip_udp_in_full_hw_cksum_err", KSTAT_DATA_UINT64 },
13969         "ip_udp_in_part_hw_cksum_err", KSTAT_DATA_UINT64 },
13970         "ip_udp_in_sw_cksum_err",  KSTAT_DATA_UINT64 },
13971         "conn_in_rcvdstaddr",      KSTAT_DATA_UINT64 },
13972         "conn_in_rcvopts",         KSTAT_DATA_UINT64 },
13973         "conn_in_rcvif",          KSTAT_DATA_UINT64 },
13974         "conn_in_rcvslla",        KSTAT_DATA_UINT64 },
13975         "conn_in_rcvucrd",        KSTAT_DATA_UINT64 },
13976         "conn_in_rcvttl",         KSTAT_DATA_UINT64 },
13977         "conn_in_rcvhopopts",     KSTAT_DATA_UINT64 },
13978         "conn_in_rcvhoplimit",    KSTAT_DATA_UINT64 },
13979         "conn_in_rcvdstopts",     KSTAT_DATA_UINT64 },
13980         "conn_in_rcvrthdrdstopts", KSTAT_DATA_UINT64 },
13981         "conn_in_rcvrthdr",       KSTAT_DATA_UINT64 },
13982         "conn_in_rcvpktinfo",     KSTAT_DATA_UINT64 },
13983         "conn_in_rcvtclass",      KSTAT_DATA_UINT64 },
13984         "conn_in_timestamp",      KSTAT_DATA_UINT64 },
13985     };

13987     ksp = kstat_create_netstack("ip", 0, "ipstat", "net",

```

```

13988         KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
13989         KSTAT_FLAG_VIRTUAL, stackid);

13991     if (ksp == NULL)
13992         return (NULL);

13994     bcopy(&template, ip_statisticsp, sizeof (template));
13995     ksp->ks_data = (void *)ip_statisticsp;
13996     ksp->ks_private = (void *) (uintptr_t)stackid;

13998     kstat_install(ksp);
13999     return (ksp);
14000 }

14002 static void
14003 ip_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
14004 {
14005     if (ksp != NULL) {
14006         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
14007         kstat_delete_netstack(ksp, stackid);
14008     }
14009 }

14011 static void *
14012 ip_kstat_init(netstackid_t stackid, ip_stack_t *ipst)
14013 {
14014     kstat_t *ksp;

14016     ip_named_kstat_t template = {
14017         "forwarding",          KSTAT_DATA_UINT32, 0 },
14018         "defaultTTL",         KSTAT_DATA_UINT32, 0 },
14019         "inReceives",         KSTAT_DATA_UINT64, 0 },
14020         "inHdrErrors",        KSTAT_DATA_UINT32, 0 },
14021         "inAddrErrors",       KSTAT_DATA_UINT32, 0 },
14022         "forwDatagrams",      KSTAT_DATA_UINT64, 0 },
14023         "inUnknownProtos",    KSTAT_DATA_UINT32, 0 },
14024         "inDiscards",         KSTAT_DATA_UINT32, 0 },
14025         "inDelivers",         KSTAT_DATA_UINT64, 0 },
14026         "outRequests",        KSTAT_DATA_UINT64, 0 },
14027         "outDiscards",        KSTAT_DATA_UINT32, 0 },
14028         "outNoRoutes",        KSTAT_DATA_UINT32, 0 },
14029         "reasmTimeout",       KSTAT_DATA_UINT32, 0 },
14030         "reasmReqds",         KSTAT_DATA_UINT32, 0 },
14031         "reasmOKs",           KSTAT_DATA_UINT32, 0 },
14032         "reasmFails",         KSTAT_DATA_UINT32, 0 },
14033         "fragOKs",            KSTAT_DATA_UINT32, 0 },
14034         "fragFails",          KSTAT_DATA_UINT32, 0 },
14035         "fragCreates",        KSTAT_DATA_UINT32, 0 },
14036         "addrEntrySize",      KSTAT_DATA_INT32, 0 },
14037         "routeEntrySize",     KSTAT_DATA_INT32, 0 },
14038         "netToMediaEntrySize", KSTAT_DATA_INT32, 0 },
14039         "routingDiscards",    KSTAT_DATA_UINT32, 0 },
14040         "inErrs",             KSTAT_DATA_UINT32, 0 },
14041         "noPorts",            KSTAT_DATA_UINT32, 0 },
14042         "inCksumErrs",        KSTAT_DATA_UINT32, 0 },
14043         "reasmDuplicates",    KSTAT_DATA_UINT32, 0 },
14044         "reasmPartDups",     KSTAT_DATA_UINT32, 0 },
14045         "forwProhibits",      KSTAT_DATA_UINT32, 0 },
14046         "udpInCksumErrs",     KSTAT_DATA_UINT32, 0 },
14047         "udpInOverflows",     KSTAT_DATA_UINT32, 0 },
14048         "rawipInOverflows",   KSTAT_DATA_UINT32, 0 },
14049         "ipsecInSucceeded",   KSTAT_DATA_UINT32, 0 },
14050         "ipsecInFailed",      KSTAT_DATA_INT32, 0 },
14051         "memberEntrySize",    KSTAT_DATA_INT32, 0 },
14052         "inIPv6",             KSTAT_DATA_UINT32, 0 },
14053         "outIPv6",            KSTAT_DATA_UINT32, 0 },

```

```

14054         { "outSwitchIPv6",    KSTAT_DATA_UINT32, 0 },
14055     };

14057     ksp = kstat_create_netstack("ip", 0, "ip", "mib2", KSTAT_TYPE_NAMED,
14058     NUM_OF_FIELDS(ip_named_kstat_t), 0, stackid);
14059     if (ksp == NULL || ksp->ks_data == NULL)
14060         return (NULL);

14062     template.forwarding.value.ui32 = WE_ARE_FORWARDING(ipst) ? 1:2;
14063     template.defaultTTL.value.ui32 = (uint32_t)ipst->ips_ip_def_ttl;
14064     template.reasmTimeout.value.ui32 = ipst->ips_ip_reassembly_timeout;
14065     template.addrEntrySize.value.i32 = sizeof (mib2_ipAddrEntry_t);
14066     template.routeEntrySize.value.i32 = sizeof (mib2_ipRouteEntry_t);

14068     template.netToMediaEntrySize.value.i32 =
14069         sizeof (mib2_ipNetToMediaEntry_t);

14071     template.memberEntrySize.value.i32 = sizeof (ipv6_member_t);

14073     bcopy(&template, ksp->ks_data, sizeof (template));
14074     ksp->ks_update = ip_kstat_update;
14075     ksp->ks_private = (void *) (uintptr_t)stackid;

14077     kstat_install(ksp);
14078     return (ksp);
14079 }

14081 static void
14082 ip_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14083 {
14084     if (ksp != NULL) {
14085         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
14086         kstat_delete_netstack(ksp, stackid);
14087     }
14088 }

14090 static int
14091 ip_kstat_update(kstat_t *kp, int rw)
14092 {
14093     ip_named_kstat_t *ipkp;
14094     mib2_ipIfStatsEntry_t ipmib;
14095     ill_walk_context_t ctx;
14096     ill_t *ill;
14097     netstackid_t stackid = (zoneid_t)(uintptr_t)kp->ks_private;
14098     netstack_t *ns;
14099     ip_stack_t *ipst;

14101     if (kp == NULL || kp->ks_data == NULL)
14102         return (EIO);

14104     if (rw == KSTAT_WRITE)
14105         return (EACCES);

14107     ns = netstack_find_by_stackid(stackid);
14108     if (ns == NULL)
14109         return (-1);
14110     ipst = ns->netstack_ip;
14111     if (ipst == NULL) {
14112         netstack_rele(ns);
14113         return (-1);
14114     }
14115     ipkp = (ip_named_kstat_t *)kp->ks_data;

14117     bcopy(&ipst->ips_ip_mib, &ipmib, sizeof (ipmib));
14118     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
14119     ill = ILL_START_WALK_V4(&ctx, ipst);

```

```

14120     for (; ill != NULL; ill = ill_next(&ctx, ill))
14121         ip_mib2_add_ip_stats(&ipmib, ill->ill_ip_mib);
14122     rw_exit(&ipst->ips_ill_g_lock);

14124     ipkp->forwarding.value.ui32 = ipmib.ipIfStatsForwarding;
14125     ipkp->defaultTTL.value.ui32 = ipmib.ipIfStatsDefaultTTL;
14126     ipkp->inReceives.value.ui64 = ipmib.ipIfStatsHCInReceives;
14127     ipkp->inHdrErrors.value.ui32 = ipmib.ipIfStatsInHdrErrors;
14128     ipkp->inAddrErrors.value.ui32 = ipmib.ipIfStatsInAddrErrors;
14129     ipkp->forwDatagrams.value.ui64 = ipmib.ipIfStatsHCOutForwDatagrams;
14130     ipkp->inUnknownProtos.value.ui32 = ipmib.ipIfStatsInUnknownProtos;
14131     ipkp->inDiscards.value.ui32 = ipmib.ipIfStatsInDiscards;
14132     ipkp->inDelivers.value.ui64 = ipmib.ipIfStatsHCInDelivers;
14133     ipkp->outRequests.value.ui64 = ipmib.ipIfStatsHCOutRequests;
14134     ipkp->outDiscards.value.ui32 = ipmib.ipIfStatsOutDiscards;
14135     ipkp->outNoRoutes.value.ui32 = ipmib.ipIfStatsOutNoRoutes;
14136     ipkp->reasmTimeout.value.ui32 = ipst->ips_ip_reassembly_timeout;
14137     ipkp->reasmReqds.value.ui32 = ipmib.ipIfStatsReasmReqds;
14138     ipkp->reasmOKs.value.ui32 = ipmib.ipIfStatsReasmOKs;
14139     ipkp->reasmFails.value.ui32 = ipmib.ipIfStatsReasmFails;
14140     ipkp->fragOKs.value.ui32 = ipmib.ipIfStatsOutFragOKs;
14141     ipkp->fragFails.value.ui32 = ipmib.ipIfStatsOutFragFails;
14142     ipkp->fragCreates.value.ui32 = ipmib.ipIfStatsOutFragCreates;

14144     ipkp->routingDiscards.value.ui32 = 0;
14145     ipkp->inErrs.value.ui32 = ipmib.tcpIfStatsInErrs;
14146     ipkp->noPorts.value.ui32 = ipmib.udpIfStatsNoPorts;
14147     ipkp->inCksumErrs.value.ui32 = ipmib.ipIfStatsInCksumErrs;
14148     ipkp->reasmDuplicates.value.ui32 = ipmib.ipIfStatsReasmDuplicates;
14149     ipkp->reasmPartDups.value.ui32 = ipmib.ipIfStatsReasmPartDups;
14150     ipkp->forwProhibits.value.ui32 = ipmib.ipIfStatsForwProhibits;
14151     ipkp->udpInCksumErrs.value.ui32 = ipmib.udpIfStatsInCksumErrs;
14152     ipkp->udpInOverflows.value.ui32 = ipmib.udpIfStatsInOverflows;
14153     ipkp->rawipInOverflows.value.ui32 = ipmib.rawipIfStatsInOverflows;
14154     ipkp->ipsecInSucceeded.value.ui32 = ipmib.ipsecIfStatsInSucceeded;
14155     ipkp->ipsecInFailed.value.ui32 = ipmib.ipsecIfStatsInFailed;

14157     ipkp->inIPv6.value.ui32 = ipmib.ipIfStatsInWrongIPVersion;
14158     ipkp->outIPv6.value.ui32 = ipmib.ipIfStatsOutWrongIPVersion;
14159     ipkp->outSwitchIPv6.value.ui32 = ipmib.ipIfStatsOutSwitchIPVersion;

14161     netstack_rele(ns);

14163     return (0);
14164 }

14166 static void *
14167 icmp_kstat_init(netstackid_t stackid)
14168 {
14169     kstat_t *ksp;

14171     icmp_named_kstat_t template = {
14172         {"inMsgs", KSTAT_DATA_UINT32},
14173         {"inErrors", KSTAT_DATA_UINT32},
14174         {"inDestUnreaches", KSTAT_DATA_UINT32},
14175         {"inTimeExcds", KSTAT_DATA_UINT32},
14176         {"inParmProbs", KSTAT_DATA_UINT32},
14177         {"inSrcQuenchs", KSTAT_DATA_UINT32},
14178         {"inRedirects", KSTAT_DATA_UINT32},
14179         {"inEchos", KSTAT_DATA_UINT32},
14180         {"inEchoReps", KSTAT_DATA_UINT32},
14181         {"inTimestamps", KSTAT_DATA_UINT32},
14182         {"inTimestampReps", KSTAT_DATA_UINT32},
14183         {"inAddrMasks", KSTAT_DATA_UINT32},
14184         {"inAddrMaskReps", KSTAT_DATA_UINT32},
14185         {"outMsgs", KSTAT_DATA_UINT32},

```

```

14186         {"outErrors", KSTAT_DATA_UINT32},
14187         {"outDestUnreaches", KSTAT_DATA_UINT32},
14188         {"outTimeExcds", KSTAT_DATA_UINT32},
14189         {"outParmProbs", KSTAT_DATA_UINT32},
14190         {"outSrcQuenchs", KSTAT_DATA_UINT32},
14191         {"outRedirects", KSTAT_DATA_UINT32},
14192         {"outEchos", KSTAT_DATA_UINT32},
14193         {"outEchoReps", KSTAT_DATA_UINT32},
14194         {"outTimestamps", KSTAT_DATA_UINT32},
14195         {"outTimestampReps", KSTAT_DATA_UINT32},
14196         {"outAddrMasks", KSTAT_DATA_UINT32},
14197         {"outAddrMaskReps", KSTAT_DATA_UINT32},
14198         {"inChksumErrs", KSTAT_DATA_UINT32},
14199         {"inUnknowns", KSTAT_DATA_UINT32},
14200         {"inFragNeeded", KSTAT_DATA_UINT32},
14201         {"outFragNeeded", KSTAT_DATA_UINT32},
14202         {"outDrops", KSTAT_DATA_UINT32},
14203         {"inOverFlows", KSTAT_DATA_UINT32},
14204         {"inBadRedirects", KSTAT_DATA_UINT32},
14205     };

14207     ksp = kstat_create_netstack("ip", 0, "icmp", "mib2", KSTAT_TYPE_NAMED,
14208         NUM_OF_FIELDS(icmp_named_kstat_t), 0, stackid);
14209     if (ksp == NULL || ksp->ks_data == NULL)
14210         return (NULL);

14212     bcopy(&template, ksp->ks_data, sizeof (template));

14214     ksp->ks_update = icmp_kstat_update;
14215     ksp->ks_private = (void *) (uintptr_t) stackid;

14217     kstat_install(ksp);
14218     return (ksp);
14219 }

14221 static void
14222 icmp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
14223 {
14224     if (ksp != NULL) {
14225         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
14226         kstat_delete_netstack(ksp, stackid);
14227     }
14228 }

14230 static int
14231 icmp_kstat_update(kstat_t *kp, int rw)
14232 {
14233     icmp_named_kstat_t *icmpkp;
14234     netstackid_t stackid = (zoneid_t) (uintptr_t) kp->ks_private;
14235     netstack_t *ns;
14236     ip_stack_t *ipst;

14238     if ((kp == NULL) || (kp->ks_data == NULL))
14239         return (EIO);

14241     if (rw == KSTAT_WRITE)
14242         return (EACCES);

14244     ns = netstack_find_by_stackid(stackid);
14245     if (ns == NULL)
14246         return (-1);
14247     ipst = ns->netstack_ip;
14248     if (ipst == NULL) {
14249         netstack_rele(ns);
14250         return (-1);
14251     }

```

```

14252     icmpkp = (icmp_named_kstat_t *)kp->ks_data;
14253
14254     icmpkp->inMsgs.value.ui32 = ipst->ips_icmp_mib.icmpInMsgs;
14255     icmpkp->inErrors.value.ui32 = ipst->ips_icmp_mib.icmpInErrors;
14256     icmpkp->inDestUnreachs.value.ui32 =
14257         ipst->ips_icmp_mib.icmpInDestUnreachs;
14258     icmpkp->inTimeExcds.value.ui32 = ipst->ips_icmp_mib.icmpInTimeExcds;
14259     icmpkp->inParmProbs.value.ui32 = ipst->ips_icmp_mib.icmpInParmProbs;
14260     icmpkp->inSrcQuenchs.value.ui32 = ipst->ips_icmp_mib.icmpInSrcQuenchs;
14261     icmpkp->inRedirects.value.ui32 = ipst->ips_icmp_mib.icmpInRedirects;
14262     icmpkp->inEchos.value.ui32 = ipst->ips_icmp_mib.icmpInEchos;
14263     icmpkp->inEchoReps.value.ui32 = ipst->ips_icmp_mib.icmpInEchoReps;
14264     icmpkp->inTimestamps.value.ui32 = ipst->ips_icmp_mib.icmpInTimestamps;
14265     icmpkp->inTimestampReps.value.ui32 =
14266         ipst->ips_icmp_mib.icmpInTimestampReps;
14267     icmpkp->inAddrMasks.value.ui32 = ipst->ips_icmp_mib.icmpInAddrMasks;
14268     icmpkp->inAddrMaskReps.value.ui32 =
14269         ipst->ips_icmp_mib.icmpInAddrMaskReps;
14270     icmpkp->outMsgs.value.ui32 = ipst->ips_icmp_mib.icmpOutMsgs;
14271     icmpkp->outErrors.value.ui32 = ipst->ips_icmp_mib.icmpOutErrors;
14272     icmpkp->outDestUnreachs.value.ui32 =
14273         ipst->ips_icmp_mib.icmpOutDestUnreachs;
14274     icmpkp->outTimeExcds.value.ui32 = ipst->ips_icmp_mib.icmpOutTimeExcds;
14275     icmpkp->outParmProbs.value.ui32 = ipst->ips_icmp_mib.icmpOutParmProbs;
14276     icmpkp->outSrcQuenchs.value.ui32 =
14277         ipst->ips_icmp_mib.icmpOutSrcQuenchs;
14278     icmpkp->outRedirects.value.ui32 = ipst->ips_icmp_mib.icmpOutRedirects;
14279     icmpkp->outEchos.value.ui32 = ipst->ips_icmp_mib.icmpOutEchos;
14280     icmpkp->outEchoReps.value.ui32 = ipst->ips_icmp_mib.icmpOutEchoReps;
14281     icmpkp->outTimestamps.value.ui32 =
14282         ipst->ips_icmp_mib.icmpOutTimestamps;
14283     icmpkp->outTimestampReps.value.ui32 =
14284         ipst->ips_icmp_mib.icmpOutTimestampReps;
14285     icmpkp->outAddrMasks.value.ui32 =
14286         ipst->ips_icmp_mib.icmpOutAddrMasks;
14287     icmpkp->outAddrMaskReps.value.ui32 =
14288         ipst->ips_icmp_mib.icmpOutAddrMaskReps;
14289     icmpkp->inCksumErrs.value.ui32 = ipst->ips_icmp_mib.icmpInCksumErrs;
14290     icmpkp->inUnknowns.value.ui32 = ipst->ips_icmp_mib.icmpInUnknowns;
14291     icmpkp->inFragNeeded.value.ui32 = ipst->ips_icmp_mib.icmpInFragNeeded;
14292     icmpkp->outFragNeeded.value.ui32 =
14293         ipst->ips_icmp_mib.icmpOutFragNeeded;
14294     icmpkp->outDrops.value.ui32 = ipst->ips_icmp_mib.icmpOutDrops;
14295     icmpkp->inOverflows.value.ui32 = ipst->ips_icmp_mib.icmpInOverflows;
14296     icmpkp->inBadRedirects.value.ui32 =
14297         ipst->ips_icmp_mib.icmpInBadRedirects;
14298
14299     netstack_rele(ns);
14300     return (0);
14301 }
14302
14303 /*
14304  * This is the fanout function for raw socket opened for SCTP. Note
14305  * that it is called after SCTP checks that there is no socket which
14306  * wants a packet. Then before SCTP handles this out of the blue packet,
14307  * this function is called to see if there is any raw socket for SCTP.
14308  * If there is and it is bound to the correct address, the packet will
14309  * be sent to that socket. Note that only one raw socket can be bound to
14310  * a port. This is assured in ipcl_sctp_hash_insert();
14311  */
14312 void
14313 ip_fanout_sctp_raw(mblk_t *mp, ipha_t *ipha, ip6_t *ip6h, uint32_t ports,
14314 ip_recv_attr_t *ira)
14315 {
14316     conn_t      *connp;
14317     queue_t     *rq;

```

```

14318     boolean_t    secure;
14319     ill_t        *ill = ira->ira_ill;
14320     ip_stack_t   *ipst = ill->ill_ipst;
14321     ipsec_stack_t *ipss = ipst->ips_netstack->netstack_ipsec;
14322     sctp_stack_t *sctps = ipst->ips_netstack->netstack_sctp;
14323     iaflags_t    iraflags = ira->ira_flags;
14324     ill_t        *rill = ira->ira_rill;
14325
14326     secure = iraflags & IRAF_IPSEC_SECURE;
14327
14328     connp = ipcl_classify_raw(mp, IPPROTO_SCTP, ports, ipha, ip6h,
14329         ira, ipst);
14330     if (connp == NULL) {
14331         /*
14332          * Although raw sctp is not summed, OOB chunks must be.
14333          * Drop the packet here if the sctp checksum failed.
14334          */
14335         if (iraflags & IRAF_SCTP_CSUM_ERR) {
14336             SCTPS_BUMP_MIB(sctps, sctpChecksumError);
14337             freemsg(mp);
14338             return;
14339         }
14340         ira->ira_ill = ira->ira_rill = NULL;
14341         sctp_oob_input(mp, ira, ipst);
14342         ira->ira_ill = ill;
14343         ira->ira_rill = rill;
14344         return;
14345     }
14346     rq = connp->conn_rq;
14347     if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrl : !canputnext(rq)) {
14348         CONN_DEC_REF(connp);
14349         BUMP_MIB(ill->ill_ip_mib, rawipIfStatsInOverflows);
14350         freemsg(mp);
14351         return;
14352     }
14353     if (((iraflags & IRAF_IS_IPV4) ?
14354         CONN_INBOUND_POLICY_PRESENT(connp, ipss) :
14355         CONN_INBOUND_POLICY_PRESENT_V6(connp, ipss)) ||
14356         secure) {
14357         mp = ipsec_check_inbound_policy(mp, connp, ipha,
14358             ip6h, ira);
14359         if (mp == NULL) {
14360             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
14361             /* Note that mp is NULL */
14362             ip_drop_input("ipIfStatsInDiscards", mp, ill);
14363             CONN_DEC_REF(connp);
14364             return;
14365         }
14366     }
14367
14368     if (iraflags & IRAF_ICMP_ERROR) {
14369         (connp->conn_recvicmp)(connp, mp, NULL, ira);
14370     } else {
14371         ill_t *rill = ira->ira_rill;
14372
14373         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
14374         /* This is the SOCK_RAW, IPPROTO_SCTP case. */
14375         ira->ira_ill = ira->ira_rill = NULL;
14376         (connp->conn_rcv)(connp, mp, NULL, ira);
14377         ira->ira_ill = ill;
14378         ira->ira_rill = rill;
14379     }
14380     CONN_DEC_REF(connp);
14381 }
14382
14383 /*

```

```

14384 * Free a packet that has the link-layer dl_unitdata_req_t or fast-path
14385 * header before the ip payload.
14386 */
14387 static void
14388 ip_xmit_flowctl_drop(ill_t *ill, mblk_t *mp, boolean_t is_fp_mp, int fp_mp_len)
14389 {
14390     int len = (mp->b_wptr - mp->b_rptr);
14391     mblk_t *ip_mp;

14393     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14394     if (is_fp_mp || len != fp_mp_len) {
14395         if (len > fp_mp_len) {
14396             /*
14397              * fastpath header and ip header in the first mblk
14398              */
14399             mp->b_rptr += fp_mp_len;
14400         } else {
14401             /*
14402              * ip_xmit_attach_llhdr had to prepend an mblk to
14403              * attach the fastpath header before ip header.
14404              */
14405             ip_mp = mp->b_cont;
14406             freeb(mp);
14407             mp = ip_mp;
14408             mp->b_rptr += (fp_mp_len - len);
14409         }
14410     } else {
14411         ip_mp = mp->b_cont;
14412         freeb(mp);
14413         mp = ip_mp;
14414     }
14415     ip_drop_output("ipIfStatsOutDiscards - flow ctl", mp, ill);
14416     freemsg(mp);
14417 }

14419 /*
14420 * Normal post fragmentation function.
14421 *
14422 * Send a packet using the passed in nce. This handles both IPv4 and IPv6
14423 * using the same state machine.
14424 *
14425 * We return an error on failure. In particular we return EWOULDBLOCK
14426 * when the driver flow controls. In that case this ensures that ip_wsrv runs
14427 * (currently by canputnext failure resulting in backenabling from GLD.)
14428 * This allows the callers of conn_ip_output() to use EWOULDBLOCK as an
14429 * indication that they can flow control until ip_wsrv() tells then to restart.
14430 *
14431 * If the nce passed by caller is incomplete, this function
14432 * queues the packet and if necessary, sends ARP request and bails.
14433 * If the Neighbor Cache passed is fully resolved, we simply prepend
14434 * the link-layer header to the packet, do ipsec hw acceleration
14435 * work if necessary, and send the packet out on the wire.
14436 */
14437 /* ARGSUSED6 */
14438 int
14439 ip_xmit(mblk_t *mp, nce_t *nce, iaflags_t iaflags, uint_t pkt_len,
14440         uint32_t xmit_hint, zoneid_t szone, zoneid_t nolzid, uintptr_t *ixacookie)
14441 {
14442     queue_t      *wq;
14443     ill_t         *ill = nce->nce_ill;
14444     ip_stack_t    *ipst = ill->ill_ipst;
14445     uint64_t      delta;
14446     boolean_t     isv6 = ill->ill_isv6;
14447     boolean_t     fp_mp;
14448     ncec_t        *ncec = nce->nce_common;
14449     int64_t       now = LBOLT_FASTPATH64;

```

```

14450     boolean_t     is_probe;

14452     DTRACE_PROBE1(ip_xmit, nce_t *, nce);

14454     ASSERT(mp != NULL);
14455     ASSERT(mp->b_datap->db_type == M_DATA);
14456     ASSERT(pkt_len == msgdsize(mp));

14458     /*
14459      * If we have already been here and are coming back after ARP/ND.
14460      * the IXAF_NO_TRACE flag is set. We skip FW_HOOKS, DTRACE and ipobs
14461      * in that case since they have seen the packet when it came here
14462      * the first time.
14463      */
14464     if (iaflags & IXAF_NO_TRACE)
14465         goto sendit;

14467     if (iaflags & IXAF_IS_IPV4) {
14468         ipha_t *ipha = (ipha_t *)mp->b_rptr;

14470         ASSERT(!isv6);
14471         ASSERT(pkt_len == ntohs(((ipha_t *)mp->b_rptr)->ipha_length));
14472         if (HOOKS4_INTERESTED_PHYSICAL_OUT(ipst) &&
14473             !(iaflags & IXAF_NO_PFHOOK)) {
14474             int     error;

14476             FW_HOOKS(ipst->ips_ip4_physical_out_event,
14477                     ipst->ips_ipv4firewall_physical_out,
14478                     NULL, ill, ipha, mp, mp, 0, ipst, error);
14479             DTRACE_PROBE1(ip4_physical_out_end,
14480                           mblk_t *, mp);
14481             if (mp == NULL)
14482                 return (error);

14484             /* The length could have changed */
14485             pkt_len = msgdsize(mp);
14486         }
14487         if (ipst->ips_ip4_observe.he_interested) {
14488             /*
14489              * Note that for TX the zoneid is the sending
14490              * zone, whether or not MLP is in play.
14491              * Since the szone argument is the IP zoneid (i.e.,
14492              * zero for exclusive-IP zones) and ipobs wants
14493              * the system zoneid, we map it here.
14494              */
14495             szone = IP_REAL_ZONEID(szone, ipst);

14497             /*
14498              * On the outbound path the destination zone will be
14499              * unknown as we're sending this packet out on the
14500              * wire.
14501              */
14502             ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
14503                       ill, ipst);
14504         }
14505         DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL,
14506                  void_ip_t *, ipha, __dtrace_ipsr_ill_t *, ill,
14507                  ipha_t *, ipha, ip6_t *, NULL, int, 0);
14508     } else {
14509         ip6_t *ip6h = (ip6_t *)mp->b_rptr;

14511         ASSERT(isv6);
14512         ASSERT(pkt_len ==
14513                ntohs(((ip6_t *)mp->b_rptr)->ip6_plen) + IPV6_HDR_LEN);
14514         if (HOOKS6_INTERESTED_PHYSICAL_OUT(ipst) &&
14515             !(iaflags & IXAF_NO_PFHOOK)) {

```

```

14516         int         error;

14518         FW_HOOKS6(ipst->ips_ip6_physical_out_event,
14519         ipst->ips_ipv6firewall_physical_out,
14520         NULL, ill, ip6h, mp, mp, 0, ipst, error);
14521         DTRACE_PROBE(ip6_physical_out_end,
14522         mblk_t *, mp);
14523         if (mp == NULL)
14524             return (error);

14526         /* The length could have changed */
14527         pkt_len = msgdsize(mp);
14528     }
14529     if (ipst->ips_ip6_observe.he_interested) {
14530         /* See above */
14531         szone = IP_REAL_ZONEID(szone, ipst);

14533         ipobs_hook(mp, IPOBS_HOOK_OUTBOUND, szone, ALL_ZONES,
14534         ill, ipst);
14535     }
14536     DTRACE_IP7(send, mblk_t *, mp, conn_t *, NULL,
14537     void_ip_t *, ip6h, __dtrace_ipsr_ill_t *, ill,
14538     ipha_t *, NULL, ip6_t *, ip6h, int, 0);
14539 }

14541 sendit:
14542 /*
14543  * We check the state without a lock because the state can never
14544  * move "backwards" to initial or incomplete.
14545  */
14546 switch (ncec->ncec_state) {
14547 case ND_REACHABLE:
14548 case ND_STALE:
14549 case ND_DELAY:
14550 case ND_PROBE:
14551     mp = ip_xmit_attach_llhdr(mp, nce);
14552     if (mp == NULL) {
14553         /*
14554          * ip_xmit_attach_llhdr has increased
14555          * ipIfStatsOutDiscards and called ip_drop_output()
14556          */
14557         return (ENOBUFS);
14558     }
14559     /*
14560      * check if nce_fastpath completed and we tagged on a
14561      * copy of nce_fp_mp in ip_xmit_attach_llhdr().
14562      */
14563     fp_mp = (mp->b_datap->db_type == M_DATA);

14565     if (fp_mp &&
14566         (ill->ill_capabilities & ILL_CAPAB_DLD_DIRECT)) {
14567         ill_dld_direct_t *idd;

14569         idd = &ill->ill_dld_capab->idc_direct;
14570         /*
14571          * Send the packet directly to DLD, where it
14572          * may be queued depending on the availability
14573          * of transmit resources at the media layer.
14574          * Return value should be taken into
14575          * account and flow control the TCP.
14576          */
14577         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14578         UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14579         pkt_len);

14581         if (ixaflags & IXAF_NO_DEV_FLOW_CTL) {

```

```

14582         (void) idd->idd_tx_df(idd->idd_tx_dh, mp,
14583         (uintptr_t)xmit_hint, IP_DROP_ON_NO_DESC);
14584     } else {
14585         uintptr_t cookie;

14587         if ((cookie = idd->idd_tx_df(idd->idd_tx_dh,
14588         mp, (uintptr_t)xmit_hint, 0)) != 0) {
14589             if (ixacookie != NULL)
14590                 *ixacookie = cookie;
14591             return (EWOULDBLOCK);
14592         }
14593     }
14594 } else {
14595     wq = ill->ill_wq;

14597     if (!(ixaflags & IXAF_NO_DEV_FLOW_CTL) &&
14598         !canputnext(wq)) {
14599         if (ixacookie != NULL)
14600             *ixacookie = 0;
14601         ip_xmit_flowctl_drop(ill, mp, fp_mp,
14602         nce->nce_fp_mp != NULL ?
14603         MBLKL(nce->nce_fp_mp) : 0);
14604         return (EWOULDBLOCK);
14605     }
14606     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCOutTransmits);
14607     UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCOutOctets,
14608     pkt_len);
14609     putnext(wq, mp);
14610 }

14612 /*
14613  * The rest of this function implements Neighbor Unreachability
14614  * detection. Determine if the nce is eligible for NUD.
14615  */
14616 if (ncec->ncec_flags & NCE_F_NONUD)
14617     return (0);

14619 ASSERT(ncec->ncec_state != ND_INCOMPLETE);

14621 /*
14622  * Check for upper layer advice
14623  */
14624 if (ixaflags & IXAF_REACH_CONF) {
14625     timeout_id_t tid;

14627     /*
14628      * It should be o.k. to check the state without
14629      * a lock here, at most we lose an advice.
14630      */
14631     ncec->ncec_last = TICK_TO_MSEC(now);
14632     if (ncec->ncec_state != ND_REACHABLE) {
14633         mutex_enter(&ncec->ncec_lock);
14634         ncec->ncec_state = ND_REACHABLE;
14635         tid = ncec->ncec_timeout_id;
14636         ncec->ncec_timeout_id = 0;
14637         mutex_exit(&ncec->ncec_lock);
14638         (void)untimeout(tid);
14639         if (ip_debug > 2) {
14640             /* ipldb */
14641             pr_addr_dbg("ip_xmit: state"
14642             " for %s changed to"
14643             " REACHABLE\n", AF_INET6,
14644             &ncec->ncec_addr);
14645         }
14646     }
14647     return (0);

```



```

14648     }
14650     delta = TICK_TO_MSEC(now) - ncec->ncec_last;
14651     ipldbg(("ip_xmit: delta = %" PRId64
14652     " ill_reachable_time = %d \n", delta,
14653     ill->ill_reachable_time));
14654     if (delta > (uint64_t)ill->ill_reachable_time) {
14655         mutex_enter(&ncec->ncec_lock);
14656         switch (ncec->ncec_state) {
14657             case ND_REACHABLE:
14658                 ASSERT((ncec->ncec_flags & NCE_F_NONUD) == 0);
14659                 /* FALLTHROUGH */
14660             case ND_STALE:
14661                 /*
14662                  * ND_REACHABLE is identical to
14663                  * ND_STALE in this specific case. If
14664                  * reachable time has expired for this
14665                  * neighbor (delta is greater than
14666                  * reachable time), conceptually, the
14667                  * neighbor cache is no longer in
14668                  * REACHABLE state, but already in
14669                  * STALE state. So the correct
14670                  * transition here is to ND_DELAY.
14671                  */
14672                 ncec->ncec_state = ND_DELAY;
14673                 mutex_exit(&ncec->ncec_lock);
14674                 nce_restart_timer(ncec,
14675                 ipst->ips_delay_first_probe_time);
14676                 if (ip_debug > 3) {
14677                     /* ip2dbg */
14678                     pr_addr_dbg("ip_xmit: state"
14679                     " for %s changed to"
14680                     " DELAY\n", AF_INET6,
14681                     &ncec->ncec_addr);
14682                 }
14683                 break;
14684             case ND_DELAY:
14685             case ND_PROBE:
14686                 mutex_exit(&ncec->ncec_lock);
14687                 /* Timers have already started */
14688                 break;
14689             case ND_UNREACHABLE:
14690                 /*
14691                  * nce_timer has detected that this ncec
14692                  * is unreachable and initiated deleting
14693                  * this ncec.
14694                  * This is a harmless race where we found the
14695                  * ncec before it was deleted and have
14696                  * just sent out a packet using this
14697                  * unreachable ncec.
14698                  */
14699                 mutex_exit(&ncec->ncec_lock);
14700                 break;
14701             default:
14702                 ASSERT(0);
14703                 mutex_exit(&ncec->ncec_lock);
14704         }
14705     }
14706     return (0);
14708 case ND_INCOMPLETE:
14709     /*
14710      * the state could have changed since we didn't hold the lock.
14711      * Re-verify state under lock.
14712      */
14713     is_probe = ipmp_packet_is_probe(mp, nce->nce_ill);

```

```

14714     mutex_enter(&ncec->ncec_lock);
14715     if (NCE_ISREACHABLE(ncec)) {
14716         mutex_exit(&ncec->ncec_lock);
14717         goto sendit;
14718     }
14719     /* queue the packet */
14720     nce_queue_mp(ncec, mp, is_probe);
14721     mutex_exit(&ncec->ncec_lock);
14722     DTRACE_PROBE2(ip_xmit_incomplete,
14723     (ncec_t *), ncec, (mbld_t *), mp);
14724     return (0);
14726 case ND_INITIAL:
14727     /*
14728      * State could have changed since we didn't hold the lock, so
14729      * re-verify state.
14730      */
14731     is_probe = ipmp_packet_is_probe(mp, nce->nce_ill);
14732     mutex_enter(&ncec->ncec_lock);
14733     if (NCE_ISREACHABLE(ncec)) {
14734         mutex_exit(&ncec->ncec_lock);
14735         goto sendit;
14736     }
14737     nce_queue_mp(ncec, mp, is_probe);
14738     if (ncec->ncec_state == ND_INITIAL) {
14739         ncec->ncec_state = ND_INCOMPLETE;
14740         mutex_exit(&ncec->ncec_lock);
14741         /*
14742          * figure out the source we want to use
14743          * and resolve it.
14744          */
14745         ip_ndp_resolve(ncec);
14746     } else {
14747         mutex_exit(&ncec->ncec_lock);
14748     }
14749     return (0);
14751 case ND_UNREACHABLE:
14752     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14753     ip_drop_output("ipIfStatsOutDiscards - ND_UNREACHABLE",
14754     mp, ill);
14755     freemsg(mp);
14756     return (0);
14758 default:
14759     ASSERT(0);
14760     BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
14761     ip_drop_output("ipIfStatsOutDiscards - ND_other",
14762     mp, ill);
14763     freemsg(mp);
14764     return (ENETUNREACH);
14765 }
14766 }
14768 /*
14769  * Return B_TRUE if the buffers differ in length or content.
14770  * This is used for comparing extension header buffers.
14771  * Note that an extension header would be declared different
14772  * even if all that changed was the next header value in that header i.e.
14773  * what really changed is the next extension header.
14774  */
14775 boolean_t
14776 ip_cmpbuf(const void *abuf, uint_t alen, boolean_t b_valid, const void *bbuf,
14777     uint_t blen)
14778 {
14779     if (!b_valid)

```

```

14780         blen = 0;
14781
14782     if (alen != blen)
14783         return (B_TRUE);
14784     if (alen == 0)
14785         return (B_FALSE); /* Both zero length */
14786     return (bcmp(abuf, bbuf, alen));
14787 }
14788
14789 /*
14790  * Preallocate memory for ip_savebuf(). Returns B_TRUE if ok.
14791  * Return B_FALSE if memory allocation fails - don't change any state!
14792  */
14793 boolean_t
14794 ip_allocbuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
14795             const void *src, uint_t srclen)
14796 {
14797     void *dst;
14798
14799     if (!src_valid)
14800         srclen = 0;
14801
14802     ASSERT(*dstlenp == 0);
14803     if (src != NULL && srclen != 0) {
14804         dst = mi_alloc(srclen, BPRI_MED);
14805         if (dst == NULL)
14806             return (B_FALSE);
14807     } else {
14808         dst = NULL;
14809     }
14810     if (*dstp != NULL)
14811         mi_free(*dstp);
14812     *dstp = dst;
14813     *dstlenp = dst == NULL ? 0 : srclen;
14814     return (B_TRUE);
14815 }
14816
14817 /*
14818  * Replace what is in *dst, *dstlen with the source.
14819  * Assumes ip_allocbuf has already been called.
14820  */
14821 void
14822 ip_savebuf(void **dstp, uint_t *dstlenp, boolean_t src_valid,
14823            const void *src, uint_t srclen)
14824 {
14825     if (!src_valid)
14826         srclen = 0;
14827
14828     ASSERT(*dstlenp == srclen);
14829     if (src != NULL && srclen != 0)
14830         bcopy(src, *dstp, srclen);
14831 }
14832
14833 /*
14834  * Free the storage pointed to by the members of an ip_pkt_t.
14835  */
14836 void
14837 ip_pkt_free(ip_pkt_t *ipp)
14838 {
14839     uint_t fields = ipp->ipp_fields;
14840
14841     if (fields & IPPF_HOPOPTS) {
14842         kmem_free(ipp->ipp_hopopts, ipp->ipp_hopoptslen);
14843         ipp->ipp_hopopts = NULL;
14844         ipp->ipp_hopoptslen = 0;
14845     }

```

```

14846     if (fields & IPPF_RTHDRDSTOPTS) {
14847         kmem_free(ipp->ipp_rthdrdstopts, ipp->ipp_rthdrdstoptslen);
14848         ipp->ipp_rthdrdstopts = NULL;
14849         ipp->ipp_rthdrdstoptslen = 0;
14850     }
14851     if (fields & IPPF_DSTOPTS) {
14852         kmem_free(ipp->ipp_dstopts, ipp->ipp_dstoptslen);
14853         ipp->ipp_dstopts = NULL;
14854         ipp->ipp_dstoptslen = 0;
14855     }
14856     if (fields & IPPF_RTHDR) {
14857         kmem_free(ipp->ipp_rthdr, ipp->ipp_rthdrhlen);
14858         ipp->ipp_rthdr = NULL;
14859         ipp->ipp_rthdrhlen = 0;
14860     }
14861     if (fields & IPPF_IPV4_OPTIONS) {
14862         kmem_free(ipp->ipp_ipv4_options, ipp->ipp_ipv4_options_len);
14863         ipp->ipp_ipv4_options = NULL;
14864         ipp->ipp_ipv4_options_len = 0;
14865     }
14866     if (fields & IPPF_LABEL_V4) {
14867         kmem_free(ipp->ipp_label_v4, ipp->ipp_label_len_v4);
14868         ipp->ipp_label_v4 = NULL;
14869         ipp->ipp_label_len_v4 = 0;
14870     }
14871     if (fields & IPPF_LABEL_V6) {
14872         kmem_free(ipp->ipp_label_v6, ipp->ipp_label_len_v6);
14873         ipp->ipp_label_v6 = NULL;
14874         ipp->ipp_label_len_v6 = 0;
14875     }
14876     ipp->ipp_fields &= ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14877                        IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);
14878 }
14879
14880 /*
14881  * Copy from src to dst and allocate as needed.
14882  * Returns zero or ENOMEM.
14883  *
14884  * The caller must initialize dst to zero.
14885  */
14886 int
14887 ip_pkt_copy(ip_pkt_t *src, ip_pkt_t *dst, int kmflag)
14888 {
14889     uint_t fields = src->ipp_fields;
14890
14891     /* Start with fields that don't require memory allocation */
14892     dst->ipp_fields = fields &
14893         ~(IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14894          IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6);
14895
14896     dst->ipp_addr = src->ipp_addr;
14897     dst->ipp_unicast_hops = src->ipp_unicast_hops;
14898     dst->ipp_hoplimit = src->ipp_hoplimit;
14899     dst->ipp_tclass = src->ipp_tclass;
14900     dst->ipp_type_of_service = src->ipp_type_of_service;
14901
14902     if (!(fields & (IPPF_HOPOPTS | IPPF_RTHDRDSTOPTS | IPPF_DSTOPTS |
14903                   IPPF_RTHDR | IPPF_IPV4_OPTIONS | IPPF_LABEL_V4 | IPPF_LABEL_V6)))
14904         return (0);
14905
14906     if (fields & IPPF_HOPOPTS) {
14907         dst->ipp_hopopts = kmem_alloc(src->ipp_hopoptslen, kmflag);
14908         if (dst->ipp_hopopts == NULL) {
14909             ip_pkt_free(dst);
14910             return (ENOMEM);
14911         }

```

```

14912     dst->ipp_fields |= IPPF_HOPOPTS;
14913     bcopy(src->ipp_hopopts, dst->ipp_hopopts,
14914           src->ipp_hopoptslen);
14915     dst->ipp_hopoptslen = src->ipp_hopoptslen;
14916 }
14917 if (fields & IPPF_RTHDRDSTOPTS) {
14918     dst->ipp_rthdrdstopts = kmem_alloc(src->ipp_rthdrdstoptslen,
14919                                         kmflag);
14920     if (dst->ipp_rthdrdstopts == NULL) {
14921         ip_pkt_free(dst);
14922         return (ENOMEM);
14923     }
14924     dst->ipp_fields |= IPPF_RTHDRDSTOPTS;
14925     bcopy(src->ipp_rthdrdstopts, dst->ipp_rthdrdstopts,
14926           src->ipp_rthdrdstoptslen);
14927     dst->ipp_rthdrdstoptslen = src->ipp_rthdrdstoptslen;
14928 }
14929 if (fields & IPPF_DSTOPTS) {
14930     dst->ipp_dstopts = kmem_alloc(src->ipp_dstoptslen, kmflag);
14931     if (dst->ipp_dstopts == NULL) {
14932         ip_pkt_free(dst);
14933         return (ENOMEM);
14934     }
14935     dst->ipp_fields |= IPPF_DSTOPTS;
14936     bcopy(src->ipp_dstopts, dst->ipp_dstopts,
14937           src->ipp_dstoptslen);
14938     dst->ipp_dstoptslen = src->ipp_dstoptslen;
14939 }
14940 if (fields & IPPF_RTHDR) {
14941     dst->ipp_rthdr = kmem_alloc(src->ipp_rthdrhlen, kmflag);
14942     if (dst->ipp_rthdr == NULL) {
14943         ip_pkt_free(dst);
14944         return (ENOMEM);
14945     }
14946     dst->ipp_fields |= IPPF_RTHDR;
14947     bcopy(src->ipp_rthdr, dst->ipp_rthdr,
14948           src->ipp_rthdrhlen);
14949     dst->ipp_rthdrhlen = src->ipp_rthdrhlen;
14950 }
14951 if (fields & IPPF_IPV4_OPTIONS) {
14952     dst->ipp_ipv4_options = kmem_alloc(src->ipp_ipv4_options_len,
14953                                         kmflag);
14954     if (dst->ipp_ipv4_options == NULL) {
14955         ip_pkt_free(dst);
14956         return (ENOMEM);
14957     }
14958     dst->ipp_fields |= IPPF_IPV4_OPTIONS;
14959     bcopy(src->ipp_ipv4_options, dst->ipp_ipv4_options,
14960           src->ipp_ipv4_options_len);
14961     dst->ipp_ipv4_options_len = src->ipp_ipv4_options_len;
14962 }
14963 if (fields & IPPF_LABEL_V4) {
14964     dst->ipp_label_v4 = kmem_alloc(src->ipp_label_len_v4, kmflag);
14965     if (dst->ipp_label_v4 == NULL) {
14966         ip_pkt_free(dst);
14967         return (ENOMEM);
14968     }
14969     dst->ipp_fields |= IPPF_LABEL_V4;
14970     bcopy(src->ipp_label_v4, dst->ipp_label_v4,
14971           src->ipp_label_len_v4);
14972     dst->ipp_label_len_v4 = src->ipp_label_len_v4;
14973 }
14974 if (fields & IPPF_LABEL_V6) {
14975     dst->ipp_label_v6 = kmem_alloc(src->ipp_label_len_v6, kmflag);
14976     if (dst->ipp_label_v6 == NULL) {
14977         ip_pkt_free(dst);

```

```

14978         return (ENOMEM);
14979     }
14980     dst->ipp_fields |= IPPF_LABEL_V6;
14981     bcopy(src->ipp_label_v6, dst->ipp_label_v6,
14982           src->ipp_label_len_v6);
14983     dst->ipp_label_len_v6 = src->ipp_label_len_v6;
14984 }
14985 if (fields & IPPF_FRAGHDR) {
14986     dst->ipp_fraghdr = kmem_alloc(src->ipp_fraghdrhlen, kmflag);
14987     if (dst->ipp_fraghdr == NULL) {
14988         ip_pkt_free(dst);
14989         return (ENOMEM);
14990     }
14991     dst->ipp_fields |= IPPF_FRAGHDR;
14992     bcopy(src->ipp_fraghdr, dst->ipp_fraghdr,
14993           src->ipp_fraghdrhlen);
14994     dst->ipp_fraghdrhlen = src->ipp_fraghdrhlen;
14995 }
14996 return (0);
14997 }
14999 /*
15000  * Returns INADDR_ANY if no source route
15001  */
15002 ipaddr_t
15003 ip_pkt_source_route_v4(const ip_pkt_t *ipp)
15004 {
15005     ipaddr_t      nexthop = INADDR_ANY;
15006     ipoptp_t      opts;
15007     uchar_t       *opt;
15008     uint8_t        optval;
15009     uint8_t        optlen;
15010     uint32_t       totalen;
15011
15012     if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
15013         return (INADDR_ANY);
15014
15015     totalen = ipp->ipp_ipv4_options_len;
15016     if (totalen & 0x3)
15017         return (INADDR_ANY);
15018
15019     for (optval = ipoptp_first2(&opts, totalen, ipp->ipp_ipv4_options);
15020          optval != IPOPT_EOL;
15021          optval = ipoptp_next(&opts)) {
15022         opt = opts.ipoptp_cur;
15023         switch (optval) {
15024             case IPOPT_LSRR:
15025                 if ((opts.ipoptp_flags & IPOPT_ERROR) != 0) {
15026                     break;
15027                 }
15028                 optlen = opts.ipoptp_len;
15029                 off = opt[IPOPT_OFFSET];
15030                 off--;
15031                 if (optlen < IP_ADDR_LEN ||
15032                     off > optlen - IP_ADDR_LEN) {
15033                     /* End of source route */
15034                     break;
15035                 }
15036                 bcopy((char *)opt + off, &nexthop, IP_ADDR_LEN);
15037                 if (nexthop == htonl(INADDR_LOOPBACK)) {
15038                     /* Ignore */
15039                     nexthop = INADDR_ANY;
15040                     break;
15041                 }
15042             }
15043     }

```

```

15044         break;
15045     }
15046 }
15047 return (nexthop);
15048 }

15050 /*
15051  * Reverse a source route.
15052  */
15053 void
15054 ip_pkt_source_route_reverse_v4(ip_pkt_t *ipp)
15055 {
15056     ipaddr_t      tmp;
15057     ipoptp_t      opts;
15058     uchar_t       *opt;
15059     uint8_t        optval;
15060     uint32_t       totallen;

15062     if (!(ipp->ipp_fields & IPPF_IPV4_OPTIONS))
15063         return;

15065     totallen = ipp->ipp_ipv4_options_len;
15066     if (totallen & 0x3)
15067         return;

15069     for (optval = ipoptp_first2(&opts, totallen, ipp->ipp_ipv4_options);
15070          optval != IPOPT_EOL;
15071          optval = ipoptp_next(&opts)) {
15072         uint8_t off1, off2;

15074         opt = opts.ipoptp_cur;
15075         switch (optval) {
15076             case IPOPT_SSRR:
15077             case IPOPT_LSRR:
15078                 if ((opts.ipoptp_flags & IPOPTP_ERROR) != 0) {
15079                     break;
15080                 }
15081                 off1 = IPOPT_MINOFF_SR - 1;
15082                 off2 = opt[IPOPT_OFFSET] - IP_ADDR_LEN - 1;
15083                 while (off2 > off1) {
15084                     bcopy(opt + off2, &tmp, IP_ADDR_LEN);
15085                     bcopy(opt + off1, opt + off2, IP_ADDR_LEN);
15086                     bcopy(&tmp, opt + off1, IP_ADDR_LEN);
15087                     off2 -= IP_ADDR_LEN;
15088                     off1 += IP_ADDR_LEN;
15089                 }
15090                 opt[IPOPT_OFFSET] = IPOPT_MINOFF_SR;
15091                 break;
15092             }
15093     }

15094 }

15096 /*
15097  * Returns NULL if no routing header
15098  */
15099 in6_addr_t *
15100 ip_pkt_source_route_v6(const ip_pkt_t *ipp)
15101 {
15102     in6_addr_t      *nexthop = NULL;
15103     ip6_rthdr0_t     *rthdr;

15105     if (!(ipp->ipp_fields & IPPF_RTHDR))
15106         return (NULL);

15108     rthdr = (ip6_rthdr0_t *)ipp->ipp_rthdr;
15109     if (rthdr->ip6r0_segleft == 0)

```

```

15110         return (NULL);

15112     nexthop = (in6_addr_t *)((char *)rthdr + sizeof (*rthdr));
15113     return (nexthop);
15114 }

15116 zoneid_t
15117 ip_get_zoneid_v4(ipaddr_t addr, mblk_t *mp, ip_recv_attr_t *ira,
15118                 zoneid_t lookup_zoneid)
15119 {
15120     ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
15121     ire_t            *ire;
15122     int              ire_flags = MATCH_IRE_TYPE;
15123     zoneid_t         zoneid = ALL_ZONES;

15125     if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
15126         return (ALL_ZONES);

15128     if (lookup_zoneid != ALL_ZONES)
15129         ire_flags |= MATCH_IRE_ZONEONLY;
15130     ire = ire_ftable_lookup_v4(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,
15131                                NULL, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15132     if (ire != NULL) {
15133         zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15134         ire_refrele(ire);
15135     }
15136     return (zoneid);
15137 }

15139 zoneid_t
15140 ip_get_zoneid_v6(in6_addr_t *addr, mblk_t *mp, const ill_t *ill,
15141                 ip_recv_attr_t *ira, zoneid_t lookup_zoneid)
15142 {
15143     ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
15144     ire_t            *ire;
15145     int              ire_flags = MATCH_IRE_TYPE;
15146     zoneid_t         zoneid = ALL_ZONES;

15148     if (is_system_labeled() && !tsol_can_accept_raw(mp, ira, B_FALSE))
15149         return (ALL_ZONES);

15151     if (IN6_IS_ADDR_LINKLOCAL(addr))
15152         ire_flags |= MATCH_IRE_ILL;

15154     if (lookup_zoneid != ALL_ZONES)
15155         ire_flags |= MATCH_IRE_ZONEONLY;
15156     ire = ire_ftable_lookup_v6(addr, NULL, NULL, IRE_LOCAL | IRE_LOOPBACK,
15157                                ill, lookup_zoneid, NULL, ire_flags, 0, ipst, NULL);
15158     if (ire != NULL) {
15159         zoneid = IP_REAL_ZONEID(ire->ire_zoneid, ipst);
15160         ire_refrele(ire);
15161     }
15162     return (zoneid);
15163 }

15165 /*
15166  * IP obserability hook support functions.
15167  */
15168 static void
15169 ipobs_init(ip_stack_t *ipst)
15170 {
15171     netid_t id;

15173     id = net_getnetidbynetstackid(ipst->ips_netstack->netstack_stackid);

15175     ipst->ips_ip4_observe_pr = net_protocol_lookup(id, NHF_INET);

```

```

15176     VERIFY(ipst->ips_ip4_observe_pr != NULL);

15178     ipst->ips_ip6_observe_pr = net_protocol_lookup(id, NHF_INET6);
15179     VERIFY(ipst->ips_ip6_observe_pr != NULL);
15180 }

15182 static void
15183 ipobs_fini(ip_stack_t *ipst)
15184 {
15186     VERIFY(net_protocol_release(ipst->ips_ip4_observe_pr) == 0);
15187     VERIFY(net_protocol_release(ipst->ips_ip6_observe_pr) == 0);
15188 }

15190 /*
15191  * hook_pkt_observe_t is composed in network byte order so that the
15192  * entire mblk_t chain handed into hook_run can be used as-is.
15193  * The caveat is that use of the fields, such as the zone fields,
15194  * requires conversion into host byte order first.
15195  */
15196 void
15197 ipobs_hook(mblk_t *mp, int htype, zoneid_t zsrc, zoneid_t zdst,
15198     const ill_t *ill, ip_stack_t *ipst)
15199 {
15200     hook_pkt_observe_t *hdr;
15201     uint64_t grifindex;
15202     mblk_t *imp;

15204     imp = allocb(sizeof (*hdr), BPRI_HI);
15205     if (imp == NULL)
15206         return;

15208     hdr = (hook_pkt_observe_t *)imp->b_rptr;
15209     /*
15210      * b_wptr is set to make the apparent size of the data in the mblk_t
15211      * to exclude the pointers at the end of hook_pkt_observer_t.
15212      */
15213     imp->b_wptr = imp->b_rptr + sizeof (dl_ipnetinfo_t);
15214     imp->b_cont = mp;

15216     ASSERT(DB_TYPE(mp) == M_DATA);

15218     if (IS_UNDER_IPMP(ill))
15219         grifindex = ipmp_ill_get_ipmp_ifindex(ill);
15220     else
15221         grifindex = 0;

15223     hdr->hpo_version = 1;
15224     hdr->hpo_htype = htons(htype);
15225     hdr->hpo_pktlen = htonl((ulong_t)msgdsz(mp));
15226     hdr->hpo_ifindex = htonl(ill->ill_phyint->phyint_ifindex);
15227     hdr->hpo_grifindex = htonl(grifindex);
15228     hdr->hpo_zsrc = htonl(zsrc);
15229     hdr->hpo_zdst = htonl(zdst);
15230     hdr->hpo_pkt = imp;
15231     hdr->hpo_ctx = ipst->ips_netstack;

15233     if (ill->ill_isv6) {
15234         hdr->hpo_family = AF_INET6;
15235         (void) hook_run(ipst->ips_ip6_net_data->netd_hooks,
15236             ipst->ips_ip6observing, (hook_data_t)hdr);
15237     } else {
15238         hdr->hpo_family = AF_INET;
15239         (void) hook_run(ipst->ips_ip4_net_data->netd_hooks,
15240             ipst->ips_ip4observing, (hook_data_t)hdr);
15241     }

```

```

15243     imp->b_cont = NULL;
15244     freemsg(imp);
15245 }

15247 /*
15248  * Utility routine that checks if 'v4srcp' is a valid address on underlying
15249  * interface 'ill'. If 'ipifp' is non-NULL, it's set to a held ipif
15250  * associated with 'v4srcp' on success. NOTE: if this is not called from
15251  * inside the IPSQ (ill_g_lock is not held), 'ill' may be removed from the
15252  * group during or after this lookup.
15253  */
15254 boolean_t
15255 ipif_lookup_testaddr_v4(ill_t *ill, const in_addr_t *v4srcp, ipif_t **ipifp)
15256 {
15257     ipif_t *ipif;

15259     ipif = ipif_lookup_addr_exact(*v4srcp, ill, ill->ill_ipst);
15260     if (ipif != NULL) {
15261         if (ipifp != NULL)
15262             *ipifp = ipif;
15263         else
15264             ipif_refrele(ipif);
15265         return (B_TRUE);
15266     }

15268     ipldbg(("ipif_lookup_testaddr_v4: cannot find ipif for src %x\n",
15269         *v4srcp));
15270     return (B_FALSE);
15271 }

15273 /*
15274  * Transport protocol call back function for CPU state change.
15275  */
15276 /* ARGSUSED */
15277 static int
15278 ip_tp_cpu_update(cpu_setup_t what, int id, void *arg)
15279 {
15280     processorid_t cpu_seqid;
15281     netstack_handle_t nh;
15282     netstack_t *ns;

15284     ASSERT(MUTEX_HELD(&cpu_lock));

15286     switch (what) {
15287     case CPU_CONFIG:
15288     case CPU_ON:
15289     case CPU_INIT:
15290     case CPU_CPUPART_IN:
15291         cpu_seqid = cpu[id]->cpu_seqid;
15292         netstack_next_init(&nh);
15293         while ((ns = netstack_next(&nh)) != NULL) {
15294             dccp_stack_cpu_add(ns->netstack_dccp, cpu_seqid);
15295         }
15296         #endif /* ! codereview */
15297         tcp_stack_cpu_add(ns->netstack_tcp, cpu_seqid);
15298         sctp_stack_cpu_add(ns->netstack_sctp, cpu_seqid);
15299         udp_stack_cpu_add(ns->netstack_udp, cpu_seqid);
15300         netstack_rele(ns);
15301     case CPU_UNCONFIG:
15302     case CPU_OFF:
15303     case CPU_CPUPART_OUT:
15304         /*
15305          * Nothing to do. We don't remove the per CPU stats from

```

```
15308             * the IP stack even when the CPU goes offline.
15309             */
15310             break;
15311 default:
15312             break;
15313     }
15314     return (0);
15315 }
```

new/usr/src/uts/common/inet/ip/ip_if.c

1

```
*****
533850 Sat Aug 18 10:37:14 2012
new/usr/src/uts/common/inet/ip/ip_if.c
dccp: properties
*****

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/systm.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_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_usscalar_t mac_type);
130 static int ip_ioctl_addr_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
131 mblk_t *mp, boolean_t need_up);
132 static int ip_ioctl_dstaddr_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
133 mblk_t *mp, boolean_t need_up);
134 static int ip_ioctl_slifzone_tail(ipif_t *ipif, zoneid_t zoneid,
135 queue_t *q, mblk_t *mp, boolean_t need_up);
136 static int ip_ioctl_flags_tail(ipif_t *ipif, uint64_t flags, queue_t *q,
137 mblk_t *mp);
138 static int ip_ioctl_netmask_tail(ipif_t *ipif, sin_t *sin, queue_t *q,
139 mblk_t *mp);
140 static int ip_ioctl_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_ioctl_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_ioctl_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_ioctl_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_nodf_v6intfid },
261     { DL_FDDI, IFT_FDDI, ETHERTYPE_IP, ETHERTYPE_IPV6,
262       ip_ether_v4_mapping, ip_ether_v6_mapping, ip_ether_v6intfid,
263       ip_nodf_v6intfid },
264     { DL_IB, IFT_IB, ETHERTYPE_IP, ETHERTYPE_IPV6,
265       ip_ib_v4_mapping, ip_ib_v6_mapping, ip_ib_v6intfid,
266       ip_nodf_v6intfid },
267     { DL_IPV4, IFT_IPV4, IPPROTO_ENCAP, IPPROTO_IPV6,
268       ip_mbcst_mapping, ip_mbcst_mapping, ip_ipv4_v6intfid,
269       ip_ipv4_v6destintfid },
270     { DL_IPV6, IFT_IPV6, IPPROTO_ENCAP, IPPROTO_IPV6,
271       ip_mbcst_mapping, ip_mbcst_mapping, ip_ipv6_v6intfid,
272       ip_ipv6_v6destintfid },
273     { DL_6TO4, IFT_6TO4, IPPROTO_ENCAP, IPPROTO_IPV6,
274       ip_mbcst_mapping, ip_mbcst_mapping, ip_ipv4_v6intfid,
275       ip_nodf_v6intfid },
276     { SUNW_DL_VNI, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
277       NULL, NULL, ip_nodf_v6intfid, ip_nodf_v6intfid },
278     { SUNW_DL_IPMP, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
279       NULL, NULL, ip_ipmp_v6intfid, ip_nodf_v6intfid },
280     { DL_OTHER, IFT_OTHER, ETHERTYPE_IP, ETHERTYPE_IPV6,
281       ip_ether_v4_mapping, ip_ether_v6_mapping, ip_nodf_v6intfid,
282       ip_nodf_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 (ip6_member_t));
335     SET_MIB(ill->ill_ip_mib->ipIfStatsGroupSourceEntrySize,
336     sizeof (ip6_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));
349
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->ip6IfIcmpEntrySize =
366     sizeof (mib2_ipv6IfIcmpEntry_t);
367 /*
368  * The ipIfStatsIfindex and ip6IfIcmpIndex will be assigned later
369  * after the phyint merge occurs in ipif_set_values -> ill_glist_insert
370  * -> ill_phyint_reinit
371  */
372 return (B_TRUE);
373 }

375 /*
376  * Completely vaporize a lower level tap and all associated interfaces.
377  * ill_delete is called only out of ip_close when the device control
378  * stream is being closed.
379  */
380 void
381 ill_delete(ill_t *ill)
382 {
383     ipif_t    *ipif;
384     ill_t    *prev_ill;
385     ip_stack_t    *ipst = ill->ill_ipst;

387     /*
388      * ill_delete may be forcibly entering the ipsq. The previous
389      * ioctl may not have completed and may need to be aborted.
390      * ipsq_flush takes care of it. If we don't need to enter the
391      * the ipsq forcibly, the 2nd invocation of ipsq_flush in

```

```

392     * ill_delete_tail is sufficient.
393     */
394     ipsq_flush(ill);

396     /*
397     * Nuke all interfaces. ipif_free will take down the interface,
398     * remove it from the list, and free the data structure.
399     * Walk down the ipif list and remove the logical interfaces
400     * first before removing the main ipif. We can't unplumb
401     * zeroth interface first in the case of IPv6 as update_conn_ill
402     * -> ip_ll_multireq de-references ill_ipif for checking
403     * POINTOPOINT.
404     *
405     * If ill_ipif was not properly initialized (i.e low on memory),
406     * then no interfaces to clean up. In this case just clean up the
407     * ill.
408     */
409     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
410         ipif_free(ipif);

412     /*
413     * clean out all the nce_t entries that depend on this
414     * ill for the ill_phys_addr.
415     */
416     nce_flush(ill, B_TRUE);

418     /* Clean up msgs on pending upcalls for mrouted */
419     reset_mrt_ill(ill);

421     update_conn_ill(ill, ipst);

423     /*
424     * Remove multicast references added as a result of calls to
425     * ip_join_allmulti().
426     */
427     ip_purge_allmulti(ill);

429     /*
430     * If the ill being deleted is under IPMP, boot it out of the illgrp.
431     */
432     if (IS_UNDER_IPMP(ill))
433         ipmp_ill_leave_illgrp(ill);

435     /*
436     * ill_down will arrange to blow off any IRE's dependent on this
437     * ILL, and shut down fragmentation reassembly.
438     */
439     ill_down(ill);

441     /* Let SCTP know, so that it can remove this from its list. */
442     sctp_update_ill(ill, SCTP_ILL_REMOVE);

444     /*
445     * Walk all CONNs that can have a reference on an ire or nce for this
446     * ill (we actually walk all that now have stale references).
447     */
448     ipcl_walk(conn_ixa_cleanup, (void *)B_TRUE, ipst);

450     /* With IPv6 we have dce_ifindex. Cleanup for neatness */
451     if (ill->ill_isv6)
452         dce_cleanup(ill->ill_phyint->phyint_ifindex, ipst);

454     /*
455     * If an address on this ILL is being used as a source address then
456     * clear out the pointers in other ILLs that point to this ILL.
457     */

```

```

458     rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_WRITER);
459     if (ill->ill_usesrc_grp_next != NULL) {
460         if (ill->ill_usesrc_ifindex == 0) { /* usesrc ILL ? */
461             ill_disband_usesrc_group(ill);
462         } else { /* consumer of the usesrc ILL */
463             prev_ill = ill_prev_usesrc(ill);
464             prev_ill->ill_usesrc_grp_next =
465                 ill->ill_usesrc_grp_next;
466         }
467     }
468     rw_exit(&ipst->ips_ill_g_usesrc_lock);
469 }

471 static void
472 ipif_non_duplicate(ipif_t *ipif)
473 {
474     ill_t *ill = ipif->ipif_ill;
475     mutex_enter(&ill->ill_lock);
476     if (ipif->ipif_flags & IPIF_DUPLICATE) {
477         ipif->ipif_flags &= ~IPIF_DUPLICATE;
478         ASSERT(ill->ill_ipif_dup_count > 0);
479         ill->ill_ipif_dup_count--;
480     }
481     mutex_exit(&ill->ill_lock);
482 }

484 /*
485  * ill_delete_tail is called from ip_modclose after all references
486  * to the closing ill are gone. The wait is done in ip_modclose
487  */
488 void
489 ill_delete_tail(ill_t *ill)
490 {
491     mblk_t **mpp;
492     ipif_t *ipif;
493     ip_stack_t *ipst = ill->ill_ipst;

495     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
496         ipif_non_duplicate(ipif);
497         (void) ipif_down_tail(ipif);
498     }

500     ASSERT(ill->ill_ipif_dup_count == 0);

502     /*
503     * If polling capability is enabled (which signifies direct
504     * upcall into IP and driver has ill saved as a handle),
505     * we need to make sure that unbind has completed before we
506     * let the ill disappear and driver no longer has any reference
507     * to this ill.
508     */
509     mutex_enter(&ill->ill_lock);
510     while (ill->ill_state_flags & ILL_DL_UNBIND_IN_PROGRESS)
511         cv_wait(&ill->ill_cv, &ill->ill_lock);
512     mutex_exit(&ill->ill_lock);
513     ASSERT(!(ill->ill_capabilities &
514         (ILL_CAPAB_DLD | ILL_CAPAB_DLD_POLL | ILL_CAPAB_DLD_DIRECT)));

516     if (ill->ill_net_type != IRE_LOOPBACK)
517         qprocsoff(ill->ill_rq);

519     /*
520     * We do an ipsq_flush once again now. New messages could have
521     * landed up from below (M_ERROR or M_HANGUP). Similarly 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         bcopy(&sap_addr, (char *)dst + phys_length, sizeof (sap_addr));
701     } else {
702         bcopy(&sap_addr, dst, sizeof (sap_addr));
703         if (phys_src == NULL)
704             bzero((char *)dst + sap_length, phys_length);
705         else
706             bcopy(phys_src, (char *)dst + sap_length, phys_length);
707     }
708 }
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      * ipsq_pending_mp_get. Caller will then use these values to restart
793      * the processing
794      */
795     add_mp->b_next = NULL;
796     add_mp->b_queue = q;
797     ipx->ipx_pending_mp = add_mp;
798     ipx->ipx_waitfor = waitfor;
799     mutex_exit(&ipx->ipx_lock);

801     if (connp != NULL)
802         connp->conn_oper_pending_ill = ipif->ipif_ill;

804     return (B_TRUE);
805 }

807 /*
808  * Retrieve the ipx_pending_mp and return it. There can be only 1 mp
809  * queued in the list.
810  */
811 mblk_t *
812 ipsq_pending_mp_get(ipsq_t *ipsq, conn_t **connpp)
813 {
814     mblk_t *curr = NULL;
815     ipxop_t *ipx = ipsq->ipsq_xop;

817     *connpp = NULL;
818     mutex_enter(&ipx->ipx_lock);
819     if (ipx->ipx_pending_mp == NULL) {
820         mutex_exit(&ipx->ipx_lock);
821         return (NULL);
822     }

824     /* There can be only 1 such excl message */
825     curr = ipx->ipx_pending_mp;
826     ASSERT(curr->b_next == NULL);
827     ipx->ipx_pending_ipif = NULL;
828     ipx->ipx_pending_mp = NULL;
829     ipx->ipx_waitfor = 0;
830     mutex_exit(&ipx->ipx_lock);

832     if (CONN_Q(curr->b_queue)) {
833         /*
834          * This mp did a refhold on the conn, at the start of the ioctl.
835          * So we can safely return a pointer to the conn to the caller.
836          */
837         *connpp = Q_TO_CONN(curr->b_queue);
838     } else {
839         *connpp = NULL;
840     }
841     curr->b_next = NULL;
842     curr->b_prev = NULL;
843     return (curr);
844 }

846 /*
847  * Cleanup the ioctl mp queued in ipx_pending_mp
848  * - Called in the ill_delete path
849  * - Called in the M_ERROR or M_HANGUP path on the ill.
850  * - Called in the conn close path.
851  */
852 * Returns success on finding the pending mblk associated with the ioctl or
853 * exclusive operation in progress, failure otherwise.

```

```

854  */
855 boolean_t
856 ipsq_pending_mp_cleanup(ill_t *ill, conn_t *connp)
857 {
858     mblk_t *mp;
859     ipxop_t *ipx;
860     queue_t *q;
861     ipif_t *ipif;
862     int cmd;

864     ASSERT(IAM_WRITER_ILL(ill));
865     ipx = ill->ill_phyint->phyint_ipsq->ipsq_xop;

867     mutex_enter(&ipx->ipx_lock);
868     mp = ipx->ipx_pending_mp;
869     if (connp != NULL) {
870         if (mp == NULL || mp->b_queue != CONNP_TO_WQ(connp)) {
871             /*
872              * Nothing to clean since the conn that is closing
873              * does not have a matching pending mblk in
874              * ipx_pending_mp.
875              */
876             mutex_exit(&ipx->ipx_lock);
877             return (B_FALSE);
878         }
879     } else {
880         /*
881          * A non-zero ill_error signifies we are called in the
882          * M_ERROR or M_HANGUP path and we need to unconditionally
883          * abort any current ioctl and do the corresponding cleanup.
884          * A zero ill_error means we are in the ill_delete path and
885          * we do the cleanup only if there is a pending mp.
886          */
887         if (mp == NULL && ill->ill_error == 0) {
888             mutex_exit(&ipx->ipx_lock);
889             return (B_FALSE);
890         }
891     }

893     /* Now remove from the ipx_pending_mp */
894     ipx->ipx_pending_mp = NULL;
895     ipif = ipx->ipx_pending_ipif;
896     ipx->ipx_pending_ipif = NULL;
897     ipx->ipx_waitfor = 0;
898     ipx->ipx_current_ipif = NULL;
899     cmd = ipx->ipx_current_ioctl;
900     ipx->ipx_current_ioctl = 0;
901     ipx->ipx_current_done = B_TRUE;
902     mutex_exit(&ipx->ipx_lock);

904     if (mp == NULL)
905         return (B_FALSE);

907     q = mp->b_queue;
908     mp->b_next = NULL;
909     mp->b_prev = NULL;
910     mp->b_queue = NULL;

912     if (DB_TYPE(mp) == M_IOCTL || DB_TYPE(mp) == M_IOCTLDATA) {
913         DTRACE_PROBE4(ipif_ioctl,
914             char *, "ipsq_pending_mp_cleanup",
915             int, cmd, ill_t *, ipif == NULL ? NULL : ipif->ipif_ill,
916             ipif_t *, ipif);
917         if (connp == NULL) {
918             ip_ioctl_finish(q, mp, ENXIO, NO_COPYOUT, NULL);
919         } else {

```

```

920         ip_ioctl_finish(q, mp, ENXIO, CONN_CLOSE, NULL);
921         mutex_enter(&ipif->ipif_ill->ill_lock);
922         ipif->ipif_state_flags &= ~IPIF_CHANGING;
923         mutex_exit(&ipif->ipif_ill->ill_lock);
924     }
925 } else {
926     inet_freemsg(mp);
927 }
928 return (B_TRUE);
929 }

931 /*
932  * Called in the conn close path and ill delete path
933  */
934 static void
935 ipsq_xopq_mp_cleanup(ill_t *ill, conn_t *connp)
936 {
937     ipsq_t *ipsq;
938     mblk_t *prev;
939     mblk_t *curr;
940     mblk_t *next;
941     queue_t *wq, *rq = NULL;
942     mblk_t *tmp_list = NULL;

944     ASSERT(IAM_WRITER_ILL(ill));
945     if (connp != NULL)
946         wq = CONNP_TO_WQ(connp);
947     else
948         wq = ill->ill_wq;

950     /*
951      * In the case of lo0 being unplumbed, ill_wq will be NULL. Guard
952      * against this here.
953      */
954     if (wq != NULL)
955         rq = RD(wq);

957     ipsq = ill->ill_phyint->phyint_ipsq;
958     /*
959      * Cleanup the ioctl mp's queued in ipsq_xopq_pending_mp if any.
960      * In the case of ioctl from a conn, there can be only 1 mp
961      * queued on the ipsq. If an ill is being unplumbed flush all
962      * the messages.
963      */
964     mutex_enter(&ipsq->ipsq_lock);
965     for (prev = NULL, curr = ipsq->ipsq_xopq_mthead; 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_mthead == curr);
975                 ipsq->ipsq_xopq_mthead = 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);
1160     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
1161         (void) ipif_down(ipif, NULL, NULL);
1163     ill_down(ill);
1165     /*
1166      * Walk all CONNs that can have a reference on an ire or nce for this
1167      * ill (we actually walk all that now have stale references).
1168      */
1169     ipcl_walk(conn_ixa_cleanup, (void *)B_TRUE, ill->ill_ipst);
1171     /* With IPv6 we have dce_ifindex. Cleanup for neatness */
1172     if (ill->ill_isv6)
1173         dce_cleanup(ill->ill_phyint->phyint_ifindex, ill->ill_ipst);
1175     ipsq_current_start(ill->ill_phyint->phyint_ipsq, ill->ill_ipif, 0);
1177     /*
1178      * Atomically test and add the pending mp if references are active.
1179      */
1180     mutex_enter(&ill->ill_lock);
1181     if (!ill_is_quiescent(ill)) {
1182         /* call cannot fail since 'conn_t' argument is NULL */
1183         (void) ipsq_pending_mp_add(NULL, ill->ill_ipif, ill->ill_rq,

```

```

1184         mp, ILL_DOWN);
1185         mutex_exit(&ill->ill_lock);
1186         return (B_FALSE);
1187     }
1188     mutex_exit(&ill->ill_lock);
1189     return (B_TRUE);
1190 }

1192 static void
1193 ill_down(ill_t *ill)
1194 {
1195     mblk_t *mp;
1196     ip_stack_t *ipst = ill->ill_ipst;

1198     /*
1199      * Blow off any IRES dependent on this ILL.
1200      * The caller needs to handle conn_ixa_cleanup
1201      */
1202     ill_delete_ires(ill);

1204     ire_walk_ill(0, 0, ill_downi, ill, ill);

1206     /* Remove any conn_*_ill depending on this ill */
1207     ipcl_walk(conn_cleanup_ill, (caddr_t)ill, ipst);

1209     /*
1210      * Free state for additional IRES.
1211      */
1212     mutex_enter(&ill->ill_saved_ire_lock);
1213     mp = ill->ill_saved_ire_mp;
1214     ill->ill_saved_ire_mp = NULL;
1215     ill->ill_saved_ire_cnt = 0;
1216     mutex_exit(&ill->ill_saved_ire_lock);
1217     freemsg(mp);
1218 }

1220 /*
1221  * ire_walk routine used to delete every IRE that depends on
1222  * 'ill'. (Always called as writer, and may only be called from ire_walk.)
1223  *
1224  * Note: since the routes added by the kernel are deleted separately,
1225  * this will only be 1) IRE_IF_CLONE and 2) manually added IRE_INTERFACE.
1226  *
1227  * We also remove references on ire_nce_cache entries that refer to the ill.
1228  */
1229 void
1230 ill_downi(ire_t *ire, char *ill_arg)
1231 {
1232     ill_t *ill = (ill_t *)ill_arg;
1233     nce_t *nce;

1235     mutex_enter(&ire->ire_lock);
1236     nce = ire->ire_nce_cache;
1237     if (nce != NULL && nce->nce_ill == ill)
1238         ire->ire_nce_cache = NULL;
1239     else
1240         nce = NULL;
1241     mutex_exit(&ire->ire_lock);
1242     if (nce != NULL)
1243         nce_refrele(nce);
1244     if (ire->ire_ill == ill) {
1245         /*
1246          * The existing interface binding for ire must be
1247          * deleted before trying to bind the route to another
1248          * interface. However, since we are using the contents of the
1249          * ire after ire_delete, the caller has to ensure that

```

```

1250         * CONDEMNED (deleted) ire's are not removed from the list
1251         * when ire_delete() returns. Currently ill_downi() is
1252         * only called as part of ire_walk*() routines, so that
1253         * the irb_refhold() done by ire_walk*() will ensure that
1254         * ire_delete() does not lead to ire_inactive().
1255         */
1256         ASSERT(ire->ire_bucket->irb_refcnt > 0);
1257         ire_delete(ire);
1258         if (ire->ire_unbound)
1259             ire_rebind(ire);
1260     }
1261 }

1263 /* Remove IRE_IF_CLONE on this ill */
1264 void
1265 ill_downi_if_clone(ire_t *ire, char *ill_arg)
1266 {
1267     ill_t *ill = (ill_t *)ill_arg;

1269     ASSERT(ire->ire_type & IRE_IF_CLONE);
1270     if (ire->ire_ill == ill)
1271         ire_delete(ire);
1272 }

1274 /* Consume an M_IOCACK of the fastpath probe. */
1275 void
1276 ill_fastpath_ack(ill_t *ill, mblk_t *mp)
1277 {
1278     mblk_t *mp1 = mp;

1280     /*
1281      * If this was the first attempt turn on the fastpath probing.
1282      */
1283     mutex_enter(&ill->ill_lock);
1284     if (ill->ill_dlpi_fastpath_state == IDS_INPROGRESS)
1285         ill->ill_dlpi_fastpath_state = IDS_OK;
1286     mutex_exit(&ill->ill_lock);

1288     /* Free the M_IOCACK mblk, hold on to the data */
1289     mp = mp->b_cont;
1290     freeb(mp1);
1291     if (mp == NULL)
1292         return;
1293     if (mp->b_cont != NULL)
1294         nce_fastpath_update(ill, mp);
1295     else
1296         ip0dbg(("ill_fastpath_ack: no b_cont\n"));
1297     freemsg(mp);
1298 }

1300 /*
1301  * Throw an M_IOCTL message downstream asking "do you know fastpath?"
1302  * The data portion of the request is a dl_unitdata_req_t template for
1303  * what we would send downstream in the absence of a fastpath confirmation.
1304  */
1305 int
1306 ill_fastpath_probe(ill_t *ill, mblk_t *dlur_mp)
1307 {
1308     struct iocblk *ioc;
1309     mblk_t *mp;

1311     if (dlur_mp == NULL)
1312         return (EINVAL);

1314     mutex_enter(&ill->ill_lock);
1315     switch (ill->ill_dlpi_fastpath_state) {

```



```

1316     case IDS_FAILED:
1317         /*
1318          * Driver NAKed the first fastpath ioctl - assume it doesn't
1319          * support it.
1320          */
1321         mutex_exit(&ill->ill_lock);
1322         return (ENOTSUP);
1323     case IDS_UNKNOWN:
1324         /* This is the first probe */
1325         ill->ill_dlpi_fastpath_state = IDS_INPROGRESS;
1326         break;
1327     default:
1328         break;
1329 }
1330 mutex_exit(&ill->ill_lock);

1332     if ((mp = 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 reneg)
1383 {
1384     ASSERT(IAM_WRITER_ILL(ill));

1386     if (ill->ill_dlpi_capab_state != IDCS_OK)
1387         return;

1389     ill->ill_dlpi_capab_state = reneg ? IDCS_RENEG : IDCS_RESET_SENT;

1391     ill_capability_send(ill, ill->ill_capab_reset_mp);
1392     ill->ill_capab_reset_mp = NULL;
1393     /*
1394      * We turn off all capabilities except those pertaining to
1395      * direct function call capabilities viz. ILL_CAPAB_DLD*
1396      * which will be turned off by the corresponding reset functions.
1397      */
1398     ill->ill_capabilities &= ~(ILL_CAPAB_HCKSUM | ILL_CAPAB_ZEROCOPY);
1399 }

1401 static void
1402 ill_capability_reset_alloc(ill_t *ill)
1403 {
1404     mblk_t *mp;
1405     size_t size = 0;
1406     int err;
1407     dl_capability_req_t *capb;

1409     ASSERT(IAM_WRITER_ILL(ill));
1410     ASSERT(ill->ill_capab_reset_mp == NULL);

1412     if (ILL_HCKSUM_CAPABLE(ill)) {
1413         size += sizeof (dl_capability_sub_t) +
1414                 sizeof (dl_capab_hcksum_t);
1415     }

1417     if (ill->ill_capabilities & ILL_CAPAB_ZEROCOPY) {
1418         size += sizeof (dl_capability_sub_t) +
1419                 sizeof (dl_capab_zerocopy_t);
1420     }

1422     if (ill->ill_capabilities & ILL_CAPAB_DLD) {
1423         size += sizeof (dl_capability_sub_t) +
1424                 sizeof (dl_capab_dld_t);
1425     }

1427     mp = allocb_wait(size + sizeof (dl_capability_req_t), BPRI_MED,
1428                     STR_NOSIG, &err);

1430     mp->b_datap->db_type = M_PROTO;
1431     bzero(mp->b_rptr, size + sizeof (dl_capability_req_t));

1433     capb = (dl_capability_req_t *)mp->b_rptr;
1434     capb->dl_primitive = DL_CAPABILITY_REQ;
1435     capb->dl_sub_offset = sizeof (dl_capability_req_t);
1436     capb->dl_sub_length = size;

1438     mp->b_wptr += sizeof (dl_capability_req_t);

1440     /*
1441      * Each handler fills in the corresponding dl_capability_sub_t
1442      * inside the mblk,
1443      */
1444     ill_capability_hcksum_reset_fill(ill, mp);
1445     ill_capability_zerocopy_reset_fill(ill, mp);
1446     ill_capability_dld_reset_fill(ill, mp);

```

```

1448     ill->ill_capab_reset_mp = mp;
1449 }

1451 static void
1452 ill_capability_id_ack(ill_t *ill, mblk_t *mp, dl_capability_sub_t *outers)
1453 {
1454     dl_capab_id_t *id_ic;
1455     uint_t sub_dl_cap = outers->dl_cap;
1456     dl_capability_sub_t *inners;
1457     uint8_t *capend;

1459     ASSERT(sub_dl_cap == DL_CAPAB_ID_WRAPPER);

1461     /*
1462      * Note: range checks here are not absolutely sufficient to
1463      * make us robust against malformed messages sent by drivers;
1464      * this is in keeping with the rest of IP's dlpi handling.
1465      * (Remember, it's coming from something else in the kernel
1466      * address space)
1467      */

1469     capend = (uint8_t *) (outers + 1) + outers->dl_length;
1470     if (capend > mp->b_wptr) {
1471         cmn_err(CE_WARN, "ill_capability_id_ack: "
1472             "malformed sub-capability too long for mblk");
1473         return;
1474     }

1476     id_ic = (dl_capab_id_t *) (outers + 1);

1478     if (outers->dl_length < sizeof (*id_ic) ||
1479         (inners = &id_ic->id_subcap,
1480          inners->dl_length > (outers->dl_length - sizeof (*inners)))) {
1481         cmn_err(CE_WARN, "ill_capability_id_ack: malformed "
1482             "encapsulated capab type %d too long for mblk",
1483             inners->dl_cap);
1484         return;
1485     }

1487     if (!dlcapabcheckqid(&id_ic->id_mid, ill->ill_lmod_rq)) {
1488         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_lmodrq)) {
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 queue
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 queue 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;
2113
2114     ASSERT(IAM_WRITER_ILL(ill));
2115
2116     if (ill->ill_isv6)
2117         return;
2118
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 }
2128
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;
2135
2136     ASSERT(IAM_WRITER_ILL(ill));
2137
2138     if (!(ill->ill_capabilities & ILL_CAPAB_DLD))
2139         return;
2140
2141     ill_mac_perim_enter(ill, &mph);
2142
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     }
2167
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          */
2183
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     }
2188
2189     ill->ill_capabilities &= ~ILL_CAPAB_DLD;
2190     ill_mac_perim_exit(ill, mph);
2191 }
2192
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  */
2217
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;
2230
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);
2236
2237     for (;;) {
2238         mp = ipst->ips_capab_taskq_head;
2239         while (mp != NULL) {

```

```

2240         ipst->ips_capab_taskq_head = mp->b_next;
2241         if (ipst->ips_capab_taskq_head == NULL)
2242             ipst->ips_capab_taskq_tail = NULL;
2243         mutex_exit(&ipst->ips_capab_taskq_lock);
2244         mp->b_next = NULL;

2246         VERIFY(taskq_dispatch(system_taskq,
2247             ill_capability_ack_thr, mp, TQ_SLEEP) != 0);
2248         mutex_enter(&ipst->ips_capab_taskq_lock);
2249         mp = ipst->ips_capab_taskq_head;
2250     }

2252     if (ipst->ips_capab_taskq_quit)
2253         break;
2254     CALLB_CPR_SAFE_BEGIN(&cprinfo);
2255     cv_wait(&ipst->ips_capab_taskq_cv, &ipst->ips_capab_taskq_lock);
2256     CALLB_CPR_SAFE_END(&cprinfo, &ipst->ips_capab_taskq_lock);
2257 }
2258 VERIFY(ipst->ips_capab_taskq_head == NULL);
2259 VERIFY(ipst->ips_capab_taskq_tail == NULL);
2260 CALLB_CPR_EXIT(&cprinfo);
2261 thread_exit();
2262 }

2264 /*
2265  * Consume a new-style hardware capabilities negotiation ack.
2266  * Called via taskq on receipt of DL_CAPABILITY_ACK.
2267  */
2268 static void
2269 ill_capability_ack_thr(void *arg)
2270 {
2271     mblk_t *mp = arg;
2272     dl_capability_ack_t *capp;
2273     dl_capability_sub_t *subp, *endp;
2274     ill_t *ill;
2275     boolean_t reneg;

2277     ill = (ill_t *)mp->b_prev;
2278     mp->b_prev = NULL;

2280     VERIFY(ipsq_enter(ill, B_FALSE, CUR_OP) == B_TRUE);

2282     if (ill->ill_dlpi_capab_state == IDCS_RESET_SENT ||
2283         ill->ill_dlpi_capab_state == IDCS_RENEG) {
2284         /*
2285          * We have received the ack for our DL_CAPAB reset request.
2286          * There 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         reneg = ill->ill_dlpi_capab_state == IDCS_RENEG;
2291         ill_capability_dld_disable(ill);
2292         ill->ill_dlpi_capab_state = IDCS_UNKNOWN;
2293         if (reneg)
2294             ill_capability_probe(ill);
2295         goto done;
2296     }

2298     if (ill->ill_dlpi_capab_state == IDCS_PROBE_SENT)
2299         ill->ill_dlpi_capab_state = IDCS_OK;

2301     capp = (dl_capability_ack_t *)mp->b_rptr;

2303     if (capp->dl_sub_length == 0) {
2304         /* no new-style capabilities */
2305         goto done;

```

```

2306     }

2308     /* make sure the driver supplied correct dl_sub_length */
2309     if ((sizeof (*capp) + capp->dl_sub_length) > MBLKL(mp)) {
2310         ip0dbg(("ill_capability_ack: bad DL_CAPABILITY_ACK, "
2311             "invalid dl_sub_length (%d)\n", capp->dl_sub_length));
2312         goto done;
2313     }

2315 #define SC(base, offset) (dl_capability_sub_t *)(((uchar_t *) (base)) + (offset))
2316 /*
2317  * There are sub-capabilities. Process the ones we know about.
2318  * Loop until we don't have room for another sub-cap header..
2319  */
2320     for (subp = SC(capp, capp->dl_sub_offset),
2321         endp = SC(subp, capp->dl_sub_length - sizeof (*subp));
2322         subp <= endp;
2323         subp = SC(subp, sizeof (dl_capability_sub_t) + subp->dl_length)) {

2325         switch (subp->dl_cap) {
2326             case DL_CAPAB_ID_WRAPPER:
2327                 ill_capability_id_ack(ill, mp, subp);
2328                 break;
2329             default:
2330                 ill_capability_dispatch(ill, mp, subp);
2331                 break;
2332         }
2333     }
2334 #undef SC
2335     done:
2336     inet_freemsg(mp);
2337     ill_capability_done(ill);
2338     ipsq_exit(ill->ill_phyint->phyint_ipsq);
2339 }

2341 /*
2342  * This needs to be started in a taskq thread to provide a cv_waitable
2343  * context.
2344  */
2345 void
2346 ill_capability_ack(ill_t *ill, mblk_t *mp)
2347 {
2348     ip_stack_t *ipst = ill->ill_ipst;

2350     mp->b_prev = (mbk_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 *ipfnnext;
2399     mblk_t *mp;
2400     time_t current_time = gethrtime_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     ipfnnext = ipf->ipf_hash_next;
2444     if (ipfnnext)
2445         ipfnnext->ipf_ptphn = ipf->ipf_ptphn;
2446     *ipf->ipf_ptphn = ipfnnext;
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;

2517      mp = send_icmp_head;
2518      send_icmp_head = send_icmp_head->b_next;
2519      mp->b_next = NULL;

2521      dst = ((ipha_t *)mp->b_rptr)->ipha_dst;

2523      iras.ira_flags = IRAF_IS_IPV4;
2524      /*
2525      * This will result in an incorrect ALL_ZONES zoneid
2526      * for broadcast and multicast packets, but we
2527      * don't send ICMP errors for those in any case.
2528      */
2529      iras.ira_zoneid = ipif_lookup_addr_zoneid(dst,
2530      ill, ipst);
2531      ip_drop_input("ICMP_TIME_EXCEEDED reass", mp, ill);
2532      icmp_time_exceeded(mp,
2533      ICMP_REASSEMBLY_TIME_EXCEEDED, &iras);
2534      ASSERT(!(iras.ira_flags & IRAF_IPSEC_SECURE));
2535      }
2536      /*
2537      * 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;

2556      /*
2557      * If we are here within ip_min_frag_prune_time msecs remove
2558      * ill_frag_free_num_pkts oldest packets from each bucket and increment
2559      * ill_frag_free_num_pkts.
2560      */
2561      mutex_enter(&ill->ill_lock);
2562      now = ddi_get_lbolt();
2563      if (TICK_TO_MSEC(now - ill->ill_last_frag_clean_time) <=
2564      (ip_min_frag_prune_time != 0 ?
2565      ip_min_frag_prune_time : msec_per_tick)) {

2567      ill->ill_frag_free_num_pkts++;

2569      } else {

```

```

2570      ill->ill_frag_free_num_pkts = 0;
2571      }
2572      ill->ill_last_frag_clean_time = now;
2573      mutex_exit(&ill->ill_lock);

2575      /*
2576      * free ill_frag_free_num_pkts oldest packets from each bucket.
2577      */
2578      if (ill->ill_frag_free_num_pkts != 0) {
2579      int ix;

2581      for (ix = 0; ix < ILL_FRAG_HASH_TBL_COUNT; ix++) {
2582      ipfb = &ill->ill_frag_hash_tbl[ix];
2583      mutex_enter(&ipfb->ipfb_lock);
2584      if (ipfb->ipfb_ipf != NULL) {
2585      ill_frag_free_pkts(ill, ipfb, ipfb->ipfb_ipf,
2586      ill->ill_frag_free_num_pkts);
2587      }
2588      mutex_exit(&ipfb->ipfb_lock);
2589      }
2590      }
2591      /*
2592      * While the reassembly list for this ILL is too big, prune a fragment
2593      * queue by age, oldest first.
2594      */
2595      while (ill->ill_frag_count > max_count) {
2596      int ix;
2597      ipfb_t *oipfb = NULL;
2598      uint_t oldest = UINT_MAX;

2600      count = 0;
2601      for (ix = 0; ix < ILL_FRAG_HASH_TBL_COUNT; ix++) {
2602      ipfb = &ill->ill_frag_hash_tbl[ix];
2603      mutex_enter(&ipfb->ipfb_lock);
2604      ipf = ipfb->ipfb_ipf;
2605      if (ipf != NULL && ipf->ipf_gen < oldest) {
2606      oldest = ipf->ipf_gen;
2607      oipfb = ipfb;
2608      }
2609      count += ipfb->ipfb_count;
2610      mutex_exit(&ipfb->ipfb_lock);
2611      }
2612      if (oipfb == NULL)
2613      break;

2615      if (count <= max_count)
2616      return; /* Somebody beat us to it, nothing to do */
2617      mutex_enter(&oipfb->ipfb_lock);
2618      ipf = oipfb->ipfb_ipf;
2619      if (ipf != NULL) {
2620      ill_frag_free_pkts(ill, oipfb, ipf, 1);
2621      }
2622      mutex_exit(&oipfb->ipfb_lock);
2623      }
2624      }

2626      /*
2627      * free 'free_cnt' fragmented packets starting at ipf.
2628      */
2629      void
2630      ill_frag_free_pkts(ill_t *ill, ipfb_t *ipfb, ipf_t *ipf, int free_cnt)
2631      {
2632      size_t count;
2633      mblk_t *mp;
2634      mblk_t *tmp;
2635      ipf_t **ipfp = ipf->ipf_ptphn;

```

```

2637     ASSERT(MUTEX_HELD(&ipfb->ipfb_lock));
2638     ASSERT(ipfp != NULL);
2639     ASSERT(ipf != NULL);

2641     while (ipf != NULL && free_cnt-- > 0) {
2642         count = ipf->ipf_count;
2643         mp = ipf->ipf_mp;
2644         ipf = ipf->ipf_hash_next;
2645         for (tmp = mp; tmp; tmp = tmp->b_cont) {
2646             IP_REASS_SET_START(tmp, 0);
2647             IP_REASS_SET_END(tmp, 0);
2648         }
2649         atomic_add_32(&ill->ill_frag_count, -count);
2650         ASSERT(ipfb->ipfb_count >= count);
2651         ipfb->ipfb_count -= count;
2652         ASSERT(ipfb->ipfb_frag_pkts > 0);
2653         ipfb->ipfb_frag_pkts--;
2654         BUMP_MIB(ill->ill_ip_mib, ipIfStatsReasmFails);
2655         ip_drop_input("ipIfStatsReasmFails", mp, ill);
2656         freemsg(mp);
2657     }

2659     if (ipf)
2660         ipf->ipf_ptphn = ipfp;
2661     ipfp[0] = ipf;
2662 }

2664 /*
2665  * Helper function for ill_forward_set().
2666  */
2667 static void
2668 ill_forward_set_on_ill(ill_t *ill, boolean_t enable)
2669 {
2670     ip_stack_t      *ipst = ill->ill_ipst;

2672     ASSERT(IAM_WRITER_ILL(ill) || RW_READ_HELD(&ipst->ips_ill_g_lock));

2674     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->ncec_common;
2753             mutex_enter(&ncec->ncec_lock);
2754             if (enable)
2755                 ncec->ncec_flags |= NCE_F_ISROUTER;
2756             else
2757                 ncec->ncec_flags &= ~NCE_F_ISROUTER;
2758             mutex_exit(&ncec->ncec_lock);
2759             nce_refrele(nce);
2760         }
2761     }
2762 }

2764 /*
2765  * 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                 ipldbg(("ill_alloc_ppa: ppa allocation"
3124                     " failed while switching"));
3125                 vmem_destroy(ifp->illif_ppa_arena);
3126                 ifp->illif_ppa_arena = NULL;
3127                 break;
3128             }
3129         }
3130     }
3131
3132     if (ifp->illif_ppa_arena != NULL) {
3133         if (ill->ill_ppa == UINT_MAX) {
3134             ppa = (int)(uintptr_t)vmem_alloc(ifp->illif_ppa_arena,
3135                 1, VM_NOSLEEP | VM_FIRSTFIT);
3136             if (ppa == 0)
3137                 return (EAGAIN);
3138             ill->ill_ppa = --ppa;
3139         } else {
3140             ppa = (int)(uintptr_t)vmem_xalloc(ifp->illif_ppa_arena,
3141                 1, /* size */
3142                 1, /* align/quantum */
3143                 0, /* phase */
3144                 0, /* nocross */
3145                 (void *)((uintptr_t)(ill->ill_ppa + 1), /* minaddr */
3146                     (void *)((uintptr_t)(ill->ill_ppa + 2), /* maxaddr */
3147                         VM_NOSLEEP | VM_FIRSTFIT);
3148             /*
3149              * Most likely the allocation failed because
3150              * the requested ppa was in use.
3151              */
3152             if (ppa == 0)
3153                 return (EEXIST);
3154         }
3155         return (0);
3156     }
3157
3158     /*
3159      * No arena is in use and not enough (>ill_no_arena) interfaces have
3160      * been plumbed to create one. Do a linear search to get a unused ppa.
3161      */
3162     if (ill->ill_ppa == UINT_MAX) {
3163         end = UINT_MAX - 1;

```

```

3164         start = 0;
3165     } else {
3166         end = start = ill->ill_ppa;
3167     }

3169     tmp_ill = avl_find(&ifp->illif_avl_by_ppa, (void *)&start, NULL);
3170     while (tmp_ill != NULL && tmp_ill->ill_ppa == start) {
3171         if (start++ >= end) {
3172             if (ill->ill_ppa == UINT_MAX)
3173                 return (EAGAIN);
3174             else
3175                 return (EEXIST);
3176         }
3177         tmp_ill = avl_walk(&ifp->illif_avl_by_ppa, tmp_ill, AVL_AFTER);
3178     }
3179     ill->ill_ppa = start;
3180     return (0);
3181 }

3183 /*
3184  * Insert ill into the list of configured ill's. Once this function completes,
3185  * the ill is globally visible and is available through lookups. More precisely
3186  * this happens after the caller drops the ill_g_lock.
3187  */
3188 static int
3189 ill_glist_insert(ill_t *ill, char *name, boolean_t isv6)
3190 {
3191     ill_if_t *ill_interface;
3192     avl_index_t where = 0;
3193     int error;
3194     int name_length;
3195     int index;
3196     boolean_t check_length = B_FALSE;
3197     ip_stack_t *ipst = ill->ill_ipst;

3199     ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_lock));

3201     name_length = mi_strlen(name) + 1;

3203     if (isv6)
3204         index = IP_V6_G_HEAD;
3205     else
3206         index = IP_V4_G_HEAD;

3208     ill_interface = IP_VX_ILL_G_LIST(index, ipst);
3209     /*
3210      * Search for interface type based on name
3211      */
3212     while (ill_interface != (ill_if_t *)&IP_VX_ILL_G_LIST(index, ipst)) {
3213         if ((ill_interface->illif_name_len == name_length) &&
3214             (strcmp(ill_interface->illif_name, name) == 0)) {
3215             break;
3216         }
3217         ill_interface = ill_interface->illif_next;
3218     }

3220     /*
3221      * Interface type not found, create one.
3222      */
3223     if (ill_interface == (ill_if_t *)&IP_VX_ILL_G_LIST(index, ipst)) {
3224         ill_g_head_t ghead;

3226         /*
3227          * allocate ill_if_t structure
3228          */
3229         ill_interface = (ill_if_t *)mi_zalloc(sizeof (ill_if_t));

```

```

3230         if (ill_interface == NULL) {
3231             return (ENOMEM);
3232         }

3234         (void) strcpy(ill_interface->illif_name, name);
3235         ill_interface->illif_name_len = name_length;

3237         avl_create(&ill_interface->illif_avl_by_ppa,
3238             ill_compare_ppa, sizeof (ill_t),
3239             offsetof(struct ill_s, ill_avl_byppa));

3241         /*
3242          * link the structure in the back to maintain order
3243          * of configuration for ifconfig output.
3244          */
3245         ghead = ipst->ips_ill_g_heads[index];
3246         insque(ill_interface, ghead.ill_g_list_tail);
3247     }

3249     if (ill->ill_ppa == UINT_MAX)
3250         check_length = B_TRUE;

3252     error = ill_alloc_ppa(ill_interface, ill);
3253     if (error != 0) {
3254         if (avl_numnodes(&ill_interface->illif_avl_by_ppa) == 0)
3255             ill_delete_interface_type(ill->ill_ifptr);
3256         return (error);
3257     }

3259     /*
3260      * When the ppa is choosen by the system, check that there is
3261      * enough space to insert ppa. if a specific ppa was passed in this
3262      * check is not required as the interface name passed in will have
3263      * the right ppa in it.
3264      */
3265     if (check_length) {
3266         /*
3267          * UINT_MAX - 1 should fit in 10 chars, alloc 12 chars.
3268          */
3269         char buf[sizeof (uint_t) * 3];

3271         /*
3272          * convert ppa to string to calculate the amount of space
3273          * required for it in the name.
3274          */
3275         numtos(ill->ill_ppa, buf);

3277         /* Do we have enough space to insert ppa ? */

3279         if ((mi_strlen(name) + mi_strlen(buf) + 1) > LIFNAMSIZ) {
3280             /* Free ppa and interface type struct */
3281             if (ill_interface->illif_ppa_arena != NULL) {
3282                 vmem_free(ill_interface->illif_ppa_arena,
3283                     (void *) (uintptr_t) (ill->ill_ppa+1), 1);
3284             }
3285             if (avl_numnodes(&ill_interface->illif_avl_by_ppa) == 0)
3286                 ill_delete_interface_type(ill->ill_ifptr);

3288             return (EINVAL);
3289         }
3290     }

3292     (void) sprintf(ill->ill_name, "%s%u", name, ill->ill_ppa);
3293     ill->ill_name_length = mi_strlen(ill->ill_name) + 1;

3295     (void) avl_find(&ill_interface->illif_avl_by_ppa, &ill->ill_ppa,

```

```

3296         &where);
3297     ill->ill_ifptr = ill_interface;
3298     avl_insert(&ill_interface->illif_avl_by_ppa, ill, where);

3300     ill_phyint_reinit(ill);
3301     return (0);
3302 }

3304 /* Initialize the per phyint ipsq used for serialization */
3305 static boolean_t
3306 ipsq_init(ill_t *ill, boolean_t enter)
3307 {
3308     ipsq_t *ipsq;
3309     ipxop_t *ipx;

3311     if ((ipsq = kmem_zalloc(sizeof (ipsq_t), KM_NOSLEEP)) == NULL)
3312         return (B_FALSE);

3314     ill->ill_phyint->phyint_ipsq = ipsq;
3315     ipx = ipsq->ipsq_xop = &ipsq->ipsq_ownxop;
3316     ipx->ipx_ipsq = ipsq;
3317     ipsq->ipsq_next = ipsq;
3318     ipsq->ipsq_phyint = ill->ill_phyint;
3319     mutex_init(&ipsq->ipsq_lock, NULL, MUTEX_DEFAULT, 0);
3320     mutex_init(&ipx->ipx_lock, NULL, MUTEX_DEFAULT, 0);
3321     ipsq->ipsq_ipst = ill->ill_ipst; /* No netstack_hold */
3322     if (enter) {
3323         ipx->ipx_writer = curthread;
3324         ipx->ipx_forced = B_FALSE;
3325         ipx->ipx_reentry_cnt = 1;
3326 #ifdef DEBUG
3327         ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
3328 #endif
3329     }
3330     return (B_TRUE);
3331 }

3333 /*
3334  * ill_init is called by ip_open when a device control stream is opened.
3335  * It does a few initializations, and shoots a DL_INFO_REQ message down
3336  * to the driver. The response is later picked up in ip_rput_dlpi and
3337  * used to set up default mechanisms for talking to the driver. (Always
3338  * called as writer.)
3339  *
3340  * If this function returns error, ip_open will call ip_close which in
3341  * turn will call ill_delete to clean up any memory allocated here that
3342  * is not yet freed.
3343  */
3344 int
3345 ill_init(queue_t *q, ill_t *ill)
3346 {
3347     int count;
3348     dl_info_req_t *dlir;
3349     mblk_t *info_mp;
3350     uchar_t *frag_ptr;

3352     /*
3353      * The ill is initialized to zero by mi_alloc(). In addition
3354      * some fields already contain valid values, initialized in
3355      * ip_open(), before we reach here.
3356      */
3357     mutex_init(&ill->ill_lock, NULL, MUTEX_DEFAULT, 0);
3358     mutex_init(&ill->ill_saved_ire_lock, NULL, MUTEX_DEFAULT, NULL);
3359     ill->ill_saved_ire_cnt = 0;

3361     ill->ill_rq = q;

```

```

3362     ill->ill_wq = WR(q);

3364     info_mp = allocb(MAX(sizeof (dl_info_req_t), sizeof (dl_info_ack_t)),
3365         BPRI_HI);
3366     if (info_mp == NULL)
3367         return (ENOMEM);

3369     /*
3370      * Allocate sufficient space to contain our fragment hash table and
3371      * the device name.
3372      */
3373     frag_ptr = (uchar_t *)mi_zalloc(ILL_FRAG_HASH_TBL_SIZE + 2 * LIFNAMSIZ);
3374     if (frag_ptr == NULL) {
3375         freemsg(info_mp);
3376         return (ENOMEM);
3377     }
3378     ill->ill_frag_ptr = frag_ptr;
3379     ill->ill_frag_free_num_pkts = 0;
3380     ill->ill_last_frag_clean_time = 0;
3381     ill->ill_frag_hash_tbl = (ipfb_t *)frag_ptr;
3382     ill->ill_name = (char *) (frag_ptr + ILL_FRAG_HASH_TBL_SIZE);
3383     for (count = 0; count < ILL_FRAG_HASH_TBL_COUNT; count++) {
3384         mutex_init(&ill->ill_frag_hash_tbl[count].ipfb_lock,
3385             NULL, MUTEX_DEFAULT, NULL);
3386     }

3388     ill->ill_phyint = (phyint_t *)mi_zalloc(sizeof (phyint_t));
3389     if (ill->ill_phyint == NULL) {
3390         freemsg(info_mp);
3391         mi_free(frag_ptr);
3392         return (ENOMEM);
3393     }

3395     mutex_init(&ill->ill_phyint->phyint_lock, NULL, MUTEX_DEFAULT, 0);
3396     /*
3397      * For now pretend this is a v4 ill. We need to set phyint_ill*
3398      * at this point because of the following reason. If we can't
3399      * enter the ipsq at some point and cv.wait, the writer that
3400      * wakes us up tries to locate us using the list of all phyints
3401      * in an ipsq and the ills from the phyint thru the phyint_ill*.
3402      * If we don't set it now, we risk a missed wakeup.
3403      */
3404     ill->ill_phyint->phyint_illv4 = ill;
3405     ill->ill_ppa = UINT_MAX;
3406     list_create(&ill->ill_nce, sizeof (nce_t), offsetof(nce_t, nce_node));

3408     ill_set_inputfn(ill);

3410     if (!ipsq_init(ill, B_TRUE)) {
3411         freemsg(info_mp);
3412         mi_free(frag_ptr);
3413         mi_free(ill->ill_phyint);
3414         return (ENOMEM);
3415     }

3417     ill->ill_state_flags |= ILL_LL_SUBNET_PENDING;

3419     /* Frag queue limit stuff */
3420     ill->ill_frag_count = 0;
3421     ill->ill_ipf_gen = 0;

3423     rw_init(&ill->ill_mcast_lock, NULL, RW_DEFAULT, NULL);
3424     mutex_init(&ill->ill_mcast_serializer, NULL, MUTEX_DEFAULT, NULL);
3425     ill->ill_global_timer = INFINITY;
3426     ill->ill_mcast_v1_time = ill->ill_mcast_v2_time = 0;
3427     ill->ill_mcast_v1_tset = ill->ill_mcast_v2_tset = 0;

```



```

3428     ill->ill_mcast_rv = MCAST_DEF_ROBUSTNESS;
3429     ill->ill_mcast_qi = MCAST_DEF_QUERY_INTERVAL;

3431     /*
3432     * Initialize IPv6 configuration variables. The IP module is always
3433     * opened as an IPv4 module. Instead tracking down the cases where
3434     * it switches to do ipv6, we'll just initialize the IPv6 configuration
3435     * here for convenience, this has no effect until the ill is set to do
3436     * IPv6.
3437     */
3438     ill->ill_reachable_time = ND_REACHABLE_TIME;
3439     ill->ill_xmit_count = ND_MAX_MULTICAST_SOLICIT;
3440     ill->ill_max_buf = ND_MAX_Q;
3441     ill->ill_refcnt = 0;

3443     /* Send down the Info Request to the driver. */
3444     info_mp->b_datap->db_type = M_PCPROTO;
3445     dliir = (dl_info_req_t *)info_mp->b_rptr;
3446     info_mp->b_wptr = (uchar_t *)&dliir[1];
3447     dliir->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     }
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. So dup detect can't be done
3649  * using this func.
3650  */
3651 ill_t *
3652 ill_lookup_on_name(char *name, boolean_t do_alloc, boolean_t isv6,
3653     boolean_t *did_alloc, ip_stack_t *ipst)
3654 {
3655     ill_t *ill;
3656     ipif_t *ipif;
3657     ipsq_t *ipsq;
3658     kstat_named_t *kn;
3659     boolean_t isloopback;
3660     in6_addr_t ov6addr;
3662     isloopback = mi_strcmp(name, ipif_loopback_name) == 0;
3664     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
3665     ill = ill_find_by_name(name, isv6, ipst);
3666     rw_exit(&ipst->ips_ill_g_lock);
3667     if (ill != NULL)
3668         return (ill);
3670     /*
3671      * Couldn't find it. Does this happen to be a lookup for the
3672      * loopback device and are we allowed to allocate it?
3673      */
3674     if (!isloopback || !do_alloc)
3675         return (NULL);
3677     rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
3678     ill = ill_find_by_name(name, isv6, ipst);
3679     if (ill != NULL) {
3680         rw_exit(&ipst->ips_ill_g_lock);
3681         return (ill);
3682     }
3684     /* Create the loopback device on demand */
3685     ill = (ill_t *) (mi_alloc(sizeof (ill_t) +
3686         sizeof (ipif_loopback_name), BPRI_MED));
3687     if (ill == NULL)
3688         goto done;
3690     *ill = ill_null;
3691     mutex_init(&ill->ill_lock, NULL, MUTEX_DEFAULT, NULL);

```

```

3692     ill->ill_ipst = ipst;
3693     list_create(&ill->ill_nce, sizeof (nce_t), offsetof(nce_t, nce_node));
3694     netstack_hold(ipst->ips_netstack);
3695     /*
3696      * For exclusive stacks we set the zoneid to zero
3697      * to make IP operate as if in the global zone.
3698      */
3699     ill->ill_zoneid = GLOBAL_ZONEID;

3701     ill->ill_phyint = (phyint_t *)mi_zalloc(sizeof (phyint_t));
3702     if (ill->ill_phyint == NULL)
3703         goto done;

3705     if (isv6)
3706         ill->ill_phyint->phyint_illv6 = ill;
3707     else
3708         ill->ill_phyint->phyint_illv4 = ill;
3709     mutex_init(&ill->ill_phyint->phyint_lock, NULL, MUTEX_DEFAULT, 0);
3710     phyint_flags_init(ill->ill_phyint, DL_LOOP);

3712     if (isv6) {
3713         ill->ill_isv6 = B_TRUE;
3714         ill->ill_max_frag = ip_loopback_mtu_v6plus;
3715     } else {
3716         ill->ill_max_frag = ip_loopback_mtuplus;
3717     }
3718     if (!ill_allocate_mibs(ill))
3719         goto done;
3720     ill->ill_current_frag = ill->ill_max_frag;
3721     ill->ill_mtu = ill->ill_max_frag; /* Initial value */
3722     ill->ill_mc_mtu = ill->ill_mtu;
3723     /*
3724      * ipif_loopback_name can't be pointed at directly because its used
3725      * by both the ipv4 and ipv6 interfaces. When the ill is removed
3726      * from the glist, ill_glist_delete() sets the first character of
3727      * ill_name to '\0'.
3728      */
3729     ill->ill_name = (char *)ill + sizeof (*ill);
3730     (void) strcpy(ill->ill_name, ipif_loopback_name);
3731     ill->ill_name_length = sizeof (ipif_loopback_name);
3732     /* Set ill_dlpi_pending for ipsq_current_finish() to work properly */
3733     ill->ill_dlpi_pending = DL_PRIM_INVAL;

3735     rw_init(&ill->ill_mcast_lock, NULL, RW_DEFAULT, NULL);
3736     mutex_init(&ill->ill_mcast_serializer, NULL, MUTEX_DEFAULT, NULL);
3737     ill->ill_global_timer = INFINITY;
3738     ill->ill_mcast_v1_time = ill->ill_mcast_v2_time = 0;
3739     ill->ill_mcast_v1_tset = ill->ill_mcast_v2_tset = 0;
3740     ill->ill_mcast_rv = MCAST_DEF_ROBUSTNESS;
3741     ill->ill_mcast_qi = MCAST_DEF_QUERY_INTERVAL;

3743     /* No resolver here. */
3744     ill->ill_net_type = IRE_LOOPBACK;

3746     /* Initialize the ipsq */
3747     if (!ipsq_init(ill, B_FALSE))
3748         goto done;

3750     ipif = ipif_allocate(ill, 0L, IRE_LOOPBACK, B_TRUE, B_TRUE, NULL);
3751     if (ipif == NULL)
3752         goto done;

3754     ill->ill_flags = ILLF_MULTICAST;

3756     ov6addr = ipif->ipif_v6lcl_addr;
3757     /* Set up default loopback address and mask. */

```

```

3758     if (!isv6) {
3759         ipaddr_t inaddr_loopback = htonl(INADDR_LOOPBACK);

3761         IN6_IPADDR_TO_V4MAPPED(inaddr_loopback, &ipif->ipif_v6lcl_addr);
3762         V4MASK_TO_V6(htonl(IN_CLASSA_NET), ipif->ipif_v6net_mask);
3763         V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
3764             ipif->ipif_v6subnet);
3765         ill->ill_flags |= ILLF_IPV4;
3766     } else {
3767         ipif->ipif_v6lcl_addr = ipv6_loopback;
3768         ipif->ipif_v6net_mask = ipv6_all_ones;
3769         V6_MASK_COPY(ipif->ipif_v6lcl_addr, ipif->ipif_v6net_mask,
3770             ipif->ipif_v6subnet);
3771         ill->ill_flags |= ILLF_IPV6;
3772     }

3774     /*
3775      * Chain us in at the end of the ill list. hold the ill
3776      * before we make it globally visible. 1 for the lookup.
3777      */
3778     ill->ill_refcnt = 0;
3779     ill_refhold(ill);

3781     ill->ill_frag_count = 0;
3782     ill->ill_frag_free_num_pkts = 0;
3783     ill->ill_last_frag_clean_time = 0;

3785     ipsq = ill->ill_phyint->phyint_ipsq;

3787     ill_set_inputfn(ill);

3789     if (ill_glist_insert(ill, "lo", isv6) != 0)
3790         cmn_err(CE_PANIC, "cannot insert loopback interface");

3792     /* Let Sctp know so that it can add this to its list */
3793     sctp_update_ill(ill, SCTP_ILL_INSERT);

3795     /*
3796      * We have already assigned ipif_v6lcl_addr above, but we need to
3797      * call sctp_update_ipif_addr() after SCTP_ILL_INSERT, which
3798      * requires to be after ill_glist_insert() since we need the
3799      * ill_index set. Pass on ipv6_loopback as the old address.
3800      */
3801     sctp_update_ipif_addr(ipif, ov6addr);

3803     ip_rts_newaddrmsg(RTM_CHGADDR, 0, ipif, RTSQ_DEFAULT);

3805     /*
3806      * ill_glist_insert() -> ill_phyint_reinit() may have merged IPSQs.
3807      * If so, free our original one.
3808      */
3809     if (ipsq != ill->ill_phyint->phyint_ipsq)
3810         ipsq_delete(ipsq);

3812     if (ipst->ips_loopback_ksp == NULL) {
3813         /* Export loopback interface statistics */
3814         ipst->ips_loopback_ksp = kstat_create_netstack("lo", 0,
3815             ipif_loopback_name, "net",
3816             KSTAT_TYPE_NAMED, 2, 0,
3817             ipst->ips_netstack->netstack_stackid);
3818         if (ipst->ips_loopback_ksp != NULL) {
3819             ipst->ips_loopback_ksp->ks_update =
3820                 loopback_kstat_update;
3821             kn = KSTAT_NAMED_PTR(ipst->ips_loopback_ksp);
3822             kstat_named_init(&kn[0], "ipackets", KSTAT_DATA_UINT32);
3823             kstat_named_init(&kn[1], "opackets", KSTAT_DATA_UINT32);

```

```

3824         ipst->ips_loopback_ksp->ks_private =
3825             (void *) (uintptr_t) ipst->ips_netstack->
3826                 netstack_stackid;
3827         kstat_install(ipst->ips_loopback_ksp);
3828     }
3829 }
3831 *did_alloc = B_TRUE;
3832 rw_exit(&ipst->ips_ill_g_lock);
3833 ill_nic_event_dispatch(ill, MAP_IPIF_ID(ill->ill_ipif->ipif_id),
3834     NE_PLUMB, ill->ill_name, ill->ill_name_length);
3835 return (ill);
3836 done:
3837 if (ill != NULL) {
3838     if (ill->ill_phyint != NULL) {
3839         ipsq = ill->ill_phyint->phyint_ipsq;
3840         if (ipsq != NULL) {
3841             ipsq->ipsq_phyint = NULL;
3842             ipsq_delete(ipsq);
3843         }
3844         mi_free(ill->ill_phyint);
3845     }
3846     ill_free_mib(ill);
3847     if (ill->ill_ipst != NULL)
3848         netstack_rele(ill->ill_ipst->ips_netstack);
3849     mi_free(ill);
3850 }
3851 rw_exit(&ipst->ips_ill_g_lock);
3852 return (NULL);
3853 }
3855 /*
3856  * For IPP calls - use the ip_stack_t for global stack.
3857  */
3858 ill_t *
3859 ill_lookup_on_ifindex_global_instance(uint_t index, boolean_t isv6)
3860 {
3861     ip_stack_t    *ipst;
3862     ill_t          *ill;
3864     ipst = netstack_find_by_stackid(GLOBAL_NETSTACKID)->netstack_ip;
3865     if (ipst == NULL) {
3866         cmn_err(CE_WARN, "No ip_stack_t for zoneid zero!\n");
3867         return (NULL);
3868     }
3870     ill = ill_lookup_on_ifindex(index, isv6, ipst);
3871     netstack_rele(ipst->ips_netstack);
3872     return (ill);
3873 }
3875 /*
3876  * Return a pointer to the ill which matches the index and IP version type.
3877  */
3878 ill_t *
3879 ill_lookup_on_ifindex(uint_t index, boolean_t isv6, ip_stack_t *ipst)
3880 {
3881     ill_t    *ill;
3882     phyint_t *phyi;
3884     /*
3885      * Indexes are stored in the phyint - a common structure
3886      * to both IPv4 and IPv6.
3887      */
3888     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
3889     phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,

```

```

3890         (void *) &index, NULL);
3891     if (phyi != NULL) {
3892         ill = isv6 ? phyi->phyint_illv6: phyi->phyint_illv4;
3893         if (ill != NULL) {
3894             mutex_enter(&ill->ill_lock);
3895             if (!ILL_IS_CONDEMNED(ill)) {
3896                 ill_refhold_locked(ill);
3897                 mutex_exit(&ill->ill_lock);
3898                 rw_exit(&ipst->ips_ill_g_lock);
3899                 return (ill);
3900             }
3901             mutex_exit(&ill->ill_lock);
3902         }
3903     }
3904     rw_exit(&ipst->ips_ill_g_lock);
3905     return (NULL);
3906 }
3908 /*
3909  * Verify whether or not an interface index is valid for the specified zoneid
3910  * to transmit packets.
3911  * It can be zero (meaning "reset") or an interface index assigned
3912  * to a non-VNI interface. (We don't use VNI interface to send packets.)
3913  */
3914 boolean_t
3915 ip_xmit_ifindex_valid(uint_t ifindex, zoneid_t zoneid, boolean_t isv6,
3916     ip_stack_t *ipst)
3917 {
3918     ill_t    *ill;
3920     if (ifindex == 0)
3921         return (B_TRUE);
3923     ill = ill_lookup_on_ifindex_zoneid(ifindex, zoneid, isv6, ipst);
3924     if (ill == NULL)
3925         return (B_FALSE);
3926     if (IS_VNI(ill)) {
3927         ill_refrele(ill);
3928         return (B_FALSE);
3929     }
3930     ill_refrele(ill);
3931     return (B_TRUE);
3932 }
3934 /*
3935  * Return the ifindex next in sequence after the passed in ifindex.
3936  * If there is no next ifindex for the given protocol, return 0.
3937  */
3938 uint_t
3939 ill_get_next_ifindex(uint_t index, boolean_t isv6, ip_stack_t *ipst)
3940 {
3941     phyint_t *phyi;
3942     phyint_t *phyi_initial;
3943     uint_t    ifindex;
3945     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
3947     if (index == 0) {
3948         phyi = avl_first(
3949             &ipst->ips_phyint_g_list->phyint_list_avl_by_index);
3950     } else {
3951         phyi = phyi_initial = avl_find(
3952             &ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3953             (void *) &index, NULL);
3954     }

```

```

3956     for (; phyi != NULL;
3957          phyi = avl_walk(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
3958                        phyi, AVL_AFTER)) {
3959         /*
3960          * If we're not returning the first interface in the tree
3961          * and we still haven't moved past the phyint_t that
3962          * corresponds to index, avl_walk needs to be called again
3963          */
3964         if (!((index != 0) && (phyi == phyi_initial))) {
3965             if (isv6) {
3966                 if ((phyi->phyint_illv6) &&
3967                     ILL_CAN_LOOKUP(phyi->phyint_illv6) &&
3968                     (phyi->phyint_illv6->ill_isv6 == 1))
3969                     break;
3970             } else {
3971                 if ((phyi->phyint_illv4) &&
3972                     ILL_CAN_LOOKUP(phyi->phyint_illv4) &&
3973                     (phyi->phyint_illv4->ill_isv6 == 0))
3974                     break;
3975             }
3976         }
3977     }
3979     rw_exit(&ipst->ips_ill_g_lock);
3981     if (phyi != NULL)
3982         ifindex = phyi->phyint_ifindex;
3983     else
3984         ifindex = 0;
3986     return (ifindex);
3987 }
3989 /*
3990  * Return the ifindex for the named interface.
3991  * If there is no next ifindex for the interface, return 0.
3992  */
3993 uint_t
3994 ill_get_ifindex_by_name(char *name, ip_stack_t *ipst)
3995 {
3996     phyint_t      *phyi;
3997     avl_index_t    where = 0;
3998     uint_t         ifindex;
4000     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
4002     if ((phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
4003                        name, &where)) == NULL) {
4004         rw_exit(&ipst->ips_ill_g_lock);
4005         return (0);
4006     }
4008     ifindex = phyi->phyint_ifindex;
4010     rw_exit(&ipst->ips_ill_g_lock);
4012     return (ifindex);
4013 }
4015 /*
4016  * Return the ifindex to be used by upper layer protocols for instance
4017  * for IPV6_RECVPKTINFO. If IPMP this is the one for the upper ill.
4018  */
4019 uint_t
4020 ill_get_upper_ifindex(const ill_t *ill)
4021 {

```

```

4022     if (IS_UNDER_IPMP(ill))
4023         return (ipmp_ill_get_ipmp_ifindex(ill));
4024     else
4025         return (ill->ill_phyint->phyint_ifindex);
4026 }
4029 /*
4030  * Obtain a reference to the ill. The ill_refcnt is a dynamic refcnt
4031  * that gives a running thread a reference to the ill. This reference must be
4032  * released by the thread when it is done accessing the ill and related
4033  * objects. ill_refcnt can not be used to account for static references
4034  * such as other structures pointing to an ill. Callers must generally
4035  * check whether an ill can be refheld by using ILL_CAN_LOOKUP macros
4036  * or be sure that the ill is not being deleted or changing state before
4037  * calling the refhold functions. A non-zero ill_refcnt ensures that the
4038  * ill won't change any of its critical state such as address, netmask etc.
4039  */
4040 void
4041 ill_refhold(ill_t *ill)
4042 {
4043     mutex_enter(&ill->ill_lock);
4044     ill->ill_refcnt++;
4045     ILL_TRACE_REF(ill);
4046     mutex_exit(&ill->ill_lock);
4047 }
4049 void
4050 ill_refhold_locked(ill_t *ill)
4051 {
4052     ASSERT(MUTEX_HELD(&ill->ill_lock));
4053     ill->ill_refcnt++;
4054     ILL_TRACE_REF(ill);
4055 }
4057 /* Returns true if we managed to get a refhold */
4058 boolean_t
4059 ill_check_and_refhold(ill_t *ill)
4060 {
4061     mutex_enter(&ill->ill_lock);
4062     if (!ILL_IS_CONDEMNED(ill)) {
4063         ill_refhold_locked(ill);
4064         mutex_exit(&ill->ill_lock);
4065         return (B_TRUE);
4066     }
4067     mutex_exit(&ill->ill_lock);
4068     return (B_FALSE);
4069 }
4071 /*
4072  * Must not be called while holding any locks. Otherwise if this is
4073  * the last reference to be released, there is a chance of recursive mutex
4074  * panic due to ill_refrele -> ipif_ill_refrele_tail -> qwriter_ip trying
4075  * to restart an ioctl.
4076  */
4077 void
4078 ill_refrele(ill_t *ill)
4079 {
4080     mutex_enter(&ill->ill_lock);
4081     ASSERT(ill->ill_refcnt != 0);
4082     ill->ill_refcnt--;
4083     ILL_UNTRACE_REF(ill);
4084     if (ill->ill_refcnt != 0) {
4085         /* Every ire pointing to the ill adds 1 to ill_refcnt */
4086         mutex_exit(&ill->ill_lock);
4087         return;

```

```

4088     }
4090     /* Drops the ill_lock */
4091     ipif_ill_refrele_tail(ill);
4092 }

4094 /*
4095  * Obtain a weak reference count on the ill. This reference ensures the
4096  * ill won't be freed, but the ill may change any of its critical state
4097  * such as netmask, address etc. Returns an error if the ill has started
4098  * closing.
4099  */
4100 boolean_t
4101 ill_waiter_inc(ill_t *ill)
4102 {
4103     mutex_enter(&ill->ill_lock);
4104     if (ill->ill_state_flags & ILL_CONDEMNED) {
4105         mutex_exit(&ill->ill_lock);
4106         return (B_FALSE);
4107     }
4108     ill->ill_waiters++;
4109     mutex_exit(&ill->ill_lock);
4110     return (B_TRUE);
4111 }

4113 void
4114 ill_waiter_dcr(ill_t *ill)
4115 {
4116     mutex_enter(&ill->ill_lock);
4117     ill->ill_waiters--;
4118     if (ill->ill_waiters == 0)
4119         cv_broadcast(&ill->ill_cv);
4120     mutex_exit(&ill->ill_lock);
4121 }

4123 /*
4124  * ip_ll_subnet_defaults is called when we get the DL_INFO_ACK back from the
4125  * driver. We construct best guess defaults for lower level information that
4126  * we need. If an interface is brought up without injection of any overriding
4127  * information from outside, we have to be ready to go with these defaults.
4128  * When we get the first DL_INFO_ACK (from ip_open() sending a DL_INFO_REQ)
4129  * we 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);
4499
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 }
4519
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;
4525
4526     ipif = ipif_lookup_multicast(ipst, zoneid, isv6);
4527     if (ipif == NULL)
4528         return (NULL);
4529
4530     ill = ipif->ipif_ill;
4531     ill_refhold(ill);
4532     ipif_refrele(ipif);
4533     return (ill);
4534 }
4535
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 *setsrtp)
4554 {
4555     ill_t *ill;
4556
4557     ill = ire_lookup_multi_ill_v4(group, zoneid, ipst, multirtp, setsrtp);
4558     if (ill != NULL)
4559         return (ill);
4560
4561     return (ill_lookup_multicast(ipst, zoneid, B_FALSE));
4562 }
4563
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;
4574
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);
4602
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 }
4608
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 pointpoint.
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 pointpoint.
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;
4883
4884     ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
4885
4886     if (ipif->ipif_refcnt != 0)
4887         return (B_FALSE);
4888
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     }
4894
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     }
4899
4900     return (B_TRUE);
4901 }
4902
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 }
4914
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 *dlnp;
4930
4931     ASSERT(MUTEX_HELD(&ill->ill_lock));
4932
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     }
4937
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;
4944
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;
4950
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     }
4977
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);
4983
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         dlnp = (dl_notify_ind_t *)mp->b_rptr;
4997         ASSERT(dlnp->dl_primitive == DL_NOTIFY_IND);
4998
4999         switch (dlnp->dl_notification) {
5000         case DL_NOTE_PHYS_ADDR:
5001             qwriter_ip(ill, ill->ill_rq, mp,
5002                 ill_set_phys_addr_tail, CUR_OP, B_TRUE);
5003             return;
5004         case DL_NOTE_REPLUMB:
5005             qwriter_ip(ill, ill->ill_rq, mp,
5006                 ill_replumb_tail, CUR_OP, B_TRUE);
5007             return;
5008         default:
5009             ASSERT(0);
5010             ill_refrele(ill);
5011         }

```

```

5012         break;

5014     case M_ERROR:
5015     case M_HANGUP:
5016         qwriter_ip(ill, ill->ill_rq, mp, ipif_all_down_tail, CUR_OP,
5017             B_TRUE);
5018         return;

5020     case M_IOCTL:
5021     case M_IOCDATA:
5022         qwriter_ip(ill, (connp != NULL ? CONNP_TO_WQ(connp) :
5023             ill->ill_wq), mp, ip_reprocess_ioctl, CUR_OP, B_TRUE);
5024         return;

5026     default:
5027         cmn_err(CE_PANIC, "ipif_ill_refrele_tail mp %p "
5028             "db_type %d\n", (void *)mp, mp->b_datap->db_type);
5029     }
5030     return;
5031 unlock:
5032     mutex_exit(&ipsq->ipsq_lock);
5033     mutex_exit(&ipx->ipx_lock);
5034     mutex_exit(&ill->ill_lock);
5035 }

5037 #ifdef DEBUG
5038 /* Reuse trace buffer from beginning (if reached the end) and record trace */
5039 static void
5040 th_trace_rrecord(th_trace_t *th_trace)
5041 {
5042     tr_buf_t *tr_buf;
5043     uint_t lastref;

5045     lastref = th_trace->th_trace_lastref;
5046     lastref++;
5047     if (lastref == TR_BUF_MAX)
5048         lastref = 0;
5049     th_trace->th_trace_lastref = lastref;
5050     tr_buf = &th_trace->th_trbuf[lastref];
5051     tr_buf->tr_time = ddi_get_lbolt();
5052     tr_buf->tr_depth = getpcstack(tr_buf->tr_stack, TR_STACK_DEPTH);
5053 }

5055 static void
5056 th_trace_free(void *value)
5057 {
5058     th_trace_t *th_trace = value;

5060     ASSERT(th_trace->th_refcnt == 0);
5061     kmem_free(th_trace, sizeof (*th_trace));
5062 }

5064 /*
5065  * Find or create the per-thread hash table used to track object references.
5066  * The ipst argument is NULL if we shouldn't allocate.
5067  *
5068  * Accesses per-thread data, so there's no need to lock here.
5069  */
5070 static mod_hash_t *
5071 th_trace_gethash(ip_stack_t *ipst)
5072 {
5073     th_hash_t *thh;

5075     if ((thh = tsd_get(ip_thread_data)) == NULL && ipst != NULL) {
5076         mod_hash_t *mh;
5077         char name[256];

```

```

5078         size_t objsize, rshift;
5079         int retv;

5081         if ((thh = kmem_alloc(sizeof (*thh), KM_NOSLEEP)) == NULL)
5082             return (NULL);
5083         (void) snprintf(name, sizeof (name), "th_trace_%p",
5084             (void *)curthread);

5086         /*
5087          * We use mod_hash_create_extended here rather than the more
5088          * obvious mod_hash_create_ptrhash because the latter has a
5089          * hard-coded KM_SLEEP, and we'd prefer to fail rather than
5090          * block.
5091          */
5092         objsize = MAX(MAX(sizeof (ill_t), sizeof (ipif_t)),
5093             MAX(sizeof (ire_t), sizeof (ncec_t)));
5094         rshift = highbit(objsize);
5095         mh = mod_hash_create_extended(name, 64, mod_hash_null_keydtor,
5096             th_trace_free, mod_hash_byptr, (void *)rshift,
5097             mod_hash_ptrkey_cmp, KM_NOSLEEP);
5098         if (mh == NULL) {
5099             kmem_free(thh, sizeof (*thh));
5100             return (NULL);
5101         }
5102         thh->thh_hash = mh;
5103         thh->thh_ipst = ipst;
5104         /*
5105          * We trace ills, ipifs, ires, and nces. All of these are
5106          * per-IP-stack, so the lock on the thread list is as well.
5107          */
5108         rw_enter(&ip_thread_rwlock, RW_WRITER);
5109         list_insert_tail(&ip_thread_list, thh);
5110         rw_exit(&ip_thread_rwlock);
5111         retv = tsd_set(ip_thread_data, thh);
5112         ASSERT(retv == 0);
5113     }
5114     return (thh != NULL ? thh->thh_hash : NULL);
5115 }

5117 boolean_t
5118 th_trace_ref(const void *obj, ip_stack_t *ipst)
5119 {
5120     th_trace_t *th_trace;
5121     mod_hash_t *mh;
5122     mod_hash_val_t val;

5124     if ((mh = th_trace_gethash(ipst)) == NULL)
5125         return (B_FALSE);

5127     /*
5128      * Attempt to locate the trace buffer for this obj and thread.
5129      * If it does not exist, then allocate a new trace buffer and
5130      * insert into the hash.
5131      */
5132     if (mod_hash_find(mh, (mod_hash_key_t)obj, &val) == MH_ERR_NOTFOUND) {
5133         th_trace = kmem_zalloc(sizeof (th_trace_t), KM_NOSLEEP);
5134         if (th_trace == NULL)
5135             return (B_FALSE);

5137         th_trace->th_id = curthread;
5138         if (mod_hash_insert(mh, (mod_hash_key_t)obj,
5139             (mod_hash_val_t)th_trace) != 0) {
5140             kmem_free(th_trace, sizeof (th_trace_t));
5141             return (B_FALSE);
5142         }
5143     } else {

```

```

5144         th_trace = (th_trace_t *)val;
5145     }
5147     ASSERT(th_trace->th_refcnt >= 0 &&
5148         th_trace->th_refcnt < TR_BUF_MAX - 1);
5150     th_trace->th_refcnt++;
5151     th_trace_rrecord(th_trace);
5152     return (B_TRUE);
5153 }
5155 /*
5156  * For the purpose of tracing a reference release, we assume that global
5157  * tracing is always on and that the same thread initiated the reference hold
5158  * is releasing.
5159  */
5160 void
5161 th_trace_unref(const void *obj)
5162 {
5163     int retv;
5164     mod_hash_t *mh;
5165     th_trace_t *th_trace;
5166     mod_hash_val_t val;
5168     mh = th_trace_gethash(NULL);
5169     retv = mod_hash_find(mh, (mod_hash_key_t)obj, &val);
5170     ASSERT(retv == 0);
5171     th_trace = (th_trace_t *)val;
5173     ASSERT(th_trace->th_refcnt > 0);
5174     th_trace->th_refcnt--;
5175     th_trace_rrecord(th_trace);
5176 }
5178 /*
5179  * If tracing has been disabled, then we assume that the reference counts are
5180  * now useless, and we clear them out before destroying the entries.
5181  */
5182 void
5183 th_trace_cleanup(const void *obj, boolean_t trace_disable)
5184 {
5185     th_hash_t *thh;
5186     mod_hash_t *mh;
5187     mod_hash_val_t val;
5188     th_trace_t *th_trace;
5189     int retv;
5191     rw_enter(&ip_thread_rwlock, RW_READER);
5192     for (thh = list_head(&ip_thread_list); thh != NULL;
5193         thh = list_next(&ip_thread_list, thh)) {
5194         if (mod_hash_find(mh = thh->thh_hash, (mod_hash_key_t)obj,
5195             &val) == 0) {
5196             th_trace = (th_trace_t *)val;
5197             if (trace_disable)
5198                 th_trace->th_refcnt = 0;
5199             retv = mod_hash_destroy(mh, (mod_hash_key_t)obj);
5200             ASSERT(retv == 0);
5201         }
5202     }
5203     rw_exit(&ip_thread_rwlock);
5204 }
5206 void
5207 ipif_trace_ref(ipif_t *ipif)
5208 {
5209     ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));

```

```

5211     if (ipif->ipif_trace_disable)
5212         return;
5214     if (!th_trace_ref(ipif, ipif->ipif_ill->ill_ipst)) {
5215         ipif->ipif_trace_disable = B_TRUE;
5216         ipif_trace_cleanup(ipif);
5217     }
5218 }
5220 void
5221 ipif_untrace_ref(ipif_t *ipif)
5222 {
5223     ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
5225     if (!ipif->ipif_trace_disable)
5226         th_trace_unref(ipif);
5227 }
5229 void
5230 ill_trace_ref(ill_t *ill)
5231 {
5232     ASSERT(MUTEX_HELD(&ill->ill_lock));
5234     if (ill->ill_trace_disable)
5235         return;
5237     if (!th_trace_ref(ill, ill->ill_ipst)) {
5238         ill->ill_trace_disable = B_TRUE;
5239         ill_trace_cleanup(ill);
5240     }
5241 }
5243 void
5244 ill_untrace_ref(ill_t *ill)
5245 {
5246     ASSERT(MUTEX_HELD(&ill->ill_lock));
5248     if (!ill->ill_trace_disable)
5249         th_trace_unref(ill);
5250 }
5252 /*
5253  * Called when ipif is unplumbed or when memory alloc fails. Note that on
5254  * failure, ipif_trace_disable is set.
5255  */
5256 static void
5257 ipif_trace_cleanup(const ipif_t *ipif)
5258 {
5259     th_trace_cleanup(ipif, ipif->ipif_trace_disable);
5260 }
5262 /*
5263  * Called when ill is unplumbed or when memory alloc fails. Note that on
5264  * failure, ill_trace_disable is set.
5265  */
5266 static void
5267 ill_trace_cleanup(const ill_t *ill)
5268 {
5269     th_trace_cleanup(ill, ill->ill_trace_disable);
5270 }
5271 #endif /* DEBUG */
5273 void
5274 ipif_refhold_locked(ipif_t *ipif)
5275 {

```

```

5276     ASSERT(MUTEX_HELD(&ipif->ipif_ill->ill_lock));
5277     ipif->ipif_refcnt++;
5278     IPIF_TRACE_REF(ipif);
5279 }

5281 void
5282 ipif_refhold(ipif_t *ipif)
5283 {
5284     ill_t *ill;

5286     ill = ipif->ipif_ill;
5287     mutex_enter(&ill->ill_lock);
5288     ipif->ipif_refcnt++;
5289     IPIF_TRACE_REF(ipif);
5290     mutex_exit(&ill->ill_lock);
5291 }

5293 /*
5294  * Must not be called while holding any locks. Otherwise if this is
5295  * the last reference to be released there is a chance of recursive mutex
5296  * panic due to ipif_refrele -> ipif_ill_refrele_tail -> qwriter_ip trying
5297  * to restart an ioctl.
5298  */
5299 void
5300 ipif_refrele(ipif_t *ipif)
5301 {
5302     ill_t *ill;

5304     ill = ipif->ipif_ill;

5306     mutex_enter(&ill->ill_lock);
5307     ASSERT(ipif->ipif_refcnt != 0);
5308     ipif->ipif_refcnt--;
5309     IPIF_UNTRACE_REF(ipif);
5310     if (ipif->ipif_refcnt != 0) {
5311         mutex_exit(&ill->ill_lock);
5312         return;
5313     }

5315     /* Drops the ill_lock */
5316     ipif_ill_refrele_tail(ill);
5317 }

5319 ipif_t *
5320 ipif_get_next_ipif(ipif_t *curr, ill_t *ill)
5321 {
5322     ipif_t *ipif;

5324     mutex_enter(&ill->ill_lock);
5325     for (ipif = (curr == NULL ? ill->ill_ipif : curr->ipif_next);
5326          ipif != NULL; ipif = ipif->ipif_next) {
5327         if (IPIF_IS_CONDEMNED(ipif))
5328             continue;
5329         ipif_refhold_locked(ipif);
5330         mutex_exit(&ill->ill_lock);
5331         return (ipif);
5332     }
5333     mutex_exit(&ill->ill_lock);
5334     return (NULL);
5335 }

5337 /*
5338  * TODO: make this table extendible at run time
5339  * Return a pointer to the mac type info for 'mac_type'
5340  */
5341 static ip_m_t *

```

```

5342 ip_m_lookup(t_uscalar_t mac_type)
5343 {
5344     ip_m_t *ipm;

5346     for (ipm = ip_m_tbl; ipm < A_END(ip_m_tbl); ipm++)
5347         if (ipm->ip_m_mac_type == mac_type)
5348             return (ipm);
5349     return (NULL);
5350 }

5352 /*
5353  * Make a link layer address from the multicast IP address *addr.
5354  * To form the link layer address, invoke the ip_m_v*mapping function
5355  * associated with the link-layer type.
5356  */
5357 void
5358 ip_mcast_mapping(ill_t *ill, uchar_t *addr, uchar_t *hwaddr)
5359 {
5360     ip_m_t *ipm;

5362     if (ill->ill_net_type == IRE_IF_NORESOLVER)
5363         return;

5365     ASSERT(addr != NULL);

5367     ipm = ip_m_lookup(ill->ill_mactype);
5368     if (ipm == NULL ||
5369         (ill->ill_isv6 && ipm->ip_m_v6mapping == NULL) ||
5370         (ill->ill_isv6 && ipm->ip_m_v4mapping == NULL)) {
5371         ip0dbg(("no mapping for ill %s mactype 0x%x\n",
5372               ill->ill_name, ill->ill_mactype));
5373         return;
5374     }
5375     if (ill->ill_isv6)
5376         (*ipm->ip_m_v6mapping)(ill, addr, hwaddr);
5377     else
5378         (*ipm->ip_m_v4mapping)(ill, addr, hwaddr);
5379 }

5381 /*
5382  * Returns B_FALSE if the IPv4 netmask pointed by 'mask' is non-contiguous.
5383  * Otherwise returns B_TRUE.
5384  */
5385 /* The netmask can be verified to be contiguous with 32 shifts and or
5386  * operations. Take the contiguous mask (in host byte order) and compute
5387  * mask | mask << 1 | mask << 2 | ... | mask << 31
5388  * the result will be the same as the 'mask' for contiguous mask.
5389  */
5390 static boolean_t
5391 ip_contiguous_mask(uint32_t mask)
5392 {
5393     uint32_t m = mask;
5394     int i;

5396     for (i = 1; i < 32; i++)
5397         m |= (mask << i);

5399     return (m == mask);
5400 }

5402 /*
5403  * ip_rt_add is called to add an IPv4 route to the forwarding table.
5404  * ill is passed in to associate it with the correct interface.
5405  * If ire_arg is set, then we return the held IRE in that location.
5406  */
5407 int

```

```

5408 ip_rt_add(ipaddr_t dst_addr, ipaddr_t mask, ipaddr_t gw_addr,
5409 ipaddr_t src_addr, int flags, ill_t *ill, ire_t **ire_arg,
5410 boolean_t ioctl_msg, struct 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);

5446     /*
5447      * Get the ipif, if any, corresponding to the gw_addr
5448      * If -ifp was specified we restrict ourselves to the ill, otherwise
5449      * we match on the gateway and destination to handle unnumbered pt-pt
5450      * interfaces.
5451      */
5452     if (ill != NULL)
5453         ipif = ipif_lookup_addr(gw_addr, ill, ALL_ZONES, ipst);
5454     else
5455         ipif = ipif_lookup_interface(gw_addr, dst_addr, ipst);
5456     if (ipif != NULL) {
5457         if (IS_VNI(ipif->ipif_ill)) {
5458             ipif_refrele(ipif);
5459             return (EINVAL);
5460         }
5461     }

5463     /*
5464      * GateD will attempt to create routes with a loopback interface
5465      * address as the gateway and with RTF_GATEWAY set. We allow
5466      * these routes to be added, but create them as interface routes
5467      * since the gateway is an interface address.
5468      */
5469     if ((ipif != NULL) && (ipif->ipif_ire_type == IRE_LOOPBACK)) {
5470         flags &= ~RTF_GATEWAY;
5471         if (gw_addr == INADDR_LOOPBACK && dst_addr == INADDR_LOOPBACK &&
5472             mask == IP_HOST_MASK) {
5473             ire = ire_ftable_lookup_v4(dst_addr, 0, 0, IRE_LOOPBACK,

```

```

5474     NULL, ALL_ZONES, NULL, MATCH_IRE_TYPE, 0, ipst,
5475     NULL);
5476     if (ire != NULL) {
5477         ire_refrele(ire);
5478         ipif_refrele(ipif);
5479         return (EEXIST);
5480     }
5481     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_fhtable_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_fhtable_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_fhtable_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->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_mhead != NULL);
6232             ipx->ipx_mptail->b_next = mp;
6233         } else {
6234             ASSERT(ipx->ipx_mhead == NULL);
6235             ipx->ipx_mhead = 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_mhead != NULL);
6243             ipsq->ipsq_xopq_mptail->b_next = mp;
6244         } else {
6245             ASSERT(ipsq->ipsq_xopq_mhead == NULL);
6246             ipsq->ipsq_xopq_mhead = 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_mhead), 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_mhead.
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_mhead) != NULL) {
6322         ipx->ipx_mhead = mp->b_next;
6323         if (ipx->ipx_mhead == 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_PROBE1(ipsq_switch, (ipsq_t *), ipsq);

6348     if (ipsq->ipsq_swxop == &ipsq->ipsq_ownxop) {
6349         /*
6350          * We're switching back to our own xop, so we have two
6351          * xop's to drain/exit: our own, and the group xop
6352          * that we are leaving.
6353          *
6354          * First, pull ourselves out of the group ipsq list.
6355          * This is safe since we're writer on ill_g_lock.
6356          */
6357         ASSERT(ipsq->ipsq_xop != &ipsq->ipsq_ownxop);

6359         xopipsq = ipx->ipx_ipsq;
6360         while (xopipsq->ipsq_next != ipsq)
6361             xopipsq = xopipsq->ipsq_next;

6363         xopipsq->ipsq_next = ipsq->ipsq_next;
6364         ipsq->ipsq_next = ipsq;
6365         ipsq->ipsq_xop = ipsq->ipsq_swxop;
6366         ipsq->ipsq_swxop = NULL;

6368         /*
6369          * Second, prepare to exit the group xop. The actual
6370          * ipsq_exit() is done at the end of this function
6371          * since we cannot hold any locks across ipsq_exit().
6372          * Note that although we drop the group's ipx_lock, no
6373          * threads can proceed since we're still ipx_writer.
6374          */
6375         leftipsq = xopipsq;
6376         mutex_exit(&ipx->ipx_lock);

6378         /*
6379          * Third, set ipx to point to our own xop (which was
6380          * inactive and therefore can be entered).
6381          */
6382         ipx = ipsq->ipsq_xop;
6383         mutex_enter(&ipx->ipx_lock);
6384         ASSERT(ipx->ipx_writer == NULL);
6385         ASSERT(ipx->ipx_current_ipif == NULL);
6386     } else {
6387         /*
6388          * We're switching from our own xop to a group xop.
6389          * The requestor of the switch must ensure that the
6390          * group xop cannot go away (e.g. by ensuring the
6391          * phyint associated with the xop cannot go away).
6392          *
6393          * If we can become writer on our new xop, then we'll
6394          * do the drain. Otherwise, the current writer of our
6395          * new xop will do the drain when it exits.
6396          *
6397          * First, splice ourselves into the group IPSQ list.

```

```

6398         * This is safe since we're writer on ill_g_lock.
6399         */
6400         ASSERT(ipsq->ipsq_xop == &ipsq->ipsq_ownxop);

6402         xopipsq = ipsq->ipsq_swxop->ipx_ipsq;
6403         while (xopipsq->ipsq_next != ipsq->ipsq_swxop->ipx_ipsq)
6404             xopipsq = xopipsq->ipsq_next;

6406         xopipsq->ipsq_next = ipsq;
6407         ipsq->ipsq_next = ipsq->ipsq_swxop->ipx_ipsq;
6408         ipsq->ipsq_xop = ipsq->ipsq_swxop;
6409         ipsq->ipsq_swxop = NULL;

6411         /*
6412          * Second, exit our own xop, since it's now unused.
6413          * This is safe since we've got the only reference.
6414          */
6415         ASSERT(ipx->ipx_writer == curthread);
6416         ipx->ipx_writer = NULL;
6417         VERIFY(--ipx->ipx_reentry_cnt == 0);
6418         ipx->ipx_ipsq_queued = B_FALSE;
6419         mutex_exit(&ipx->ipx_lock);

6421         /*
6422          * Third, set ipx to point to our new xop, and check
6423          * if we can become writer on it. If we cannot, then
6424          * the current writer will drain the IPSQ group when
6425          * it exits. Our ipsq_xop is guaranteed to be stable
6426          * because we're still holding ipsq_lock.
6427          */
6428         ipx = ipsq->ipsq_xop;
6429         mutex_enter(&ipx->ipx_lock);
6430         if (ipx->ipx_writer != NULL ||
6431             ipx->ipx_current_ipif != NULL) {
6432             goto out;
6433         }
6434     }

6436     /*
6437     * Fourth, become writer on our new ipx before we continue
6438     * with the drain. Note that we never dropped ipsq_lock
6439     * above, so no other thread could've raced with us to
6440     * become writer first. Also, we're holding ipx_lock, so
6441     * no other thread can examine the ipx right now.
6442     */
6443     ASSERT(ipx->ipx_current_ipif == NULL);
6444     ASSERT(ipx->ipx_mphhead == NULL && ipx->ipx_mptail == NULL);
6445     VERIFY(ipx->ipx_reentry_cnt++ == 0);
6446     ipx->ipx_writer = curthread;
6447     ipx->ipx_forced = B_FALSE;

6448     #ifdef DEBUG
6449     ipx->ipx_depth = getpcstack(ipx->ipx_stack, IPX_STACK_DEPTH);
6450     #endif
6451 }

6453     xopipsq = ipsq;
6454     do {
6455         /*
6456          * So that other operations operate on a consistent and
6457          * complete phyint, a switch message on an IPSQ must be
6458          * handled prior to any other operations on that IPSQ.
6459          */
6460         if ((mp = xopipsq->ipsq_switch_mp) != NULL) {
6461             xopipsq->ipsq_switch_mp = NULL;
6462             ASSERT(mp->b_next == NULL);
6463             mp->b_next = (void *)xopipsq;

```

```

6464         goto out;
6465     }

6467     if ((mp = xopipsq->ipsq_xopq_mhead) != NULL) {
6468         xopipsq->ipsq_xopq_mhead = mp->b_next;
6469         if (xopipsq->ipsq_xopq_mhead == 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 DEBBUG
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  * ill 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 ||
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_mphhead and calls
6735 * the reentry point. When the list at ipx_mphhead becomes empty ipsq_exit
6736 * proceeds to dequeue the next ioctl in ipsq_xopq_mphhead 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 calluntimeout()
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     * SIOCGIFFLAGS) 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 won't be any messages on the ipsq_xopq_mhead
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_mhead; 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_mhead = 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 (ipip->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[IFNAMSIZ - 1] = '\0';
7090         name = ifr->ifr_name;
7091         ci->ci_sin = (sin_t *)&ifr->ifr_addr;
7092         ci->ci_sin6 = NULL;
7093         ci->ci_lifr = (struct lifreq *)ifr;
7094     } else {
7095         /* This a new style SIOC[GS]LIF* command */
7096         ASSERT(ipip->ipi_cmd_type == LIF_CMD);
7097         lifr = (struct lifreq *)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 (ipip->ipi_cmd == SIOCSLIFNAME) {
7111         /*
7112          * The ioctl will be failed if the ioctl comes down
7113          * an conn stream
7114          */
7115         if (ill == NULL) {
7116             /*
7117              * Not an ill queue, return EINVAL same as the
7118              * old error code.
7119              */
7120             return (ENXIO);
7121         }
7122         ipif = ill->ill_ipif;
7123         ipif_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_usersrc_lock protects ill_usersrc_grp_next, for example, some
7257  * other thread may be trying to relink the ILLs in this usersrc group
7258  * and adjusting the ill_usersrc_grp_next pointers
7259  */
7260  rw_enter(&ipst->ips_ill_g_usersrc_lock, RW_READER);
7261  if ((ill->ill_usersrc_ifindex == 0) &&
7262      (ill->ill_usersrc_grp_next != NULL)) {
7263      for (; (ill != NULL) && (ill->ill_usersrc_grp_next != ill_head);
7264          ill = ill->ill_usersrc_grp_next)
7265          numifs++;
7266  }
7267  rw_exit(&ipst->ips_ill_g_usersrc_lock);

7269  return (numifs);
7270 }

7272 /* Null values are passed in for ipif, sin, and ifreq */
7273 /* ARGSUSED */
7274 int
7275 ip_siocctl_get_ifnum(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7276 mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7277 {
7278     int *nump;
7279     conn_t *connp = Q_TO_CONN(q);

7281     ASSERT(q->q_next == NULL); /* not a valid ioctl for ip as a module */

7283     /* Existence of b_cont->b_cont checked in ip_wput_nondata */
7284     nump = (int *)mp->b_cont->b_cont->b_rptr;

7286     *nump = ip_get_numifs(connp->conn_zoneid,
7287 connp->conn_netstack->netstack_ip);
7288     ipldbg(("ip_siocctl_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_siocctl_get_lifnum(ipif_t *dummy_ipif, sin_t *dummy_sin,
7296 queue_t *q, mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7297 {
7298     struct lifnum *lifn;
7299     mblk_t *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_siocctl_get_lifnum numifs %d", lifn->lifn_count));
7320     return (0);
7321 }

```

```

7323 /* ARGSUSED */
7324 int
7325 ip_siocctl_get_ifconf(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7326 mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7327 {
7328     STRUCT_HANDLE(ifconf, ifc);
7329     mblk_t *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 ifclen;
7337     zoneid_t zoneid;
7338     ip_stack_t *ipst = CONNQ_TO_IPST(q);

7340     ASSERT(q->q_next == NULL); /* not valid ioctls for ip as a module */

7342     ipldbg(("ip_siocctl_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      mp1 = mi_copyout_alloc(q, mp,
7408          STRUCT_FGETP(ifc, ifc_buf), ifc_bufsize, B_FALSE);
7409      if (mp1 == NULL)
7410          return (ENOMEM);

7412      mp1->b_wptr = mp1->b_rptr + ifc_bufsize;
7413  }
7414  bzero(mp1->b_rptr, mp1->b_wptr - mp1->b_rptr);
7415  /*
7416   * the SIOCGIFCONF ioctl only knows about
7417   * IPv4 addresses, so don't try to tell
7418   * it about interfaces with IPv6-only
7419   * addresses. (Last parm 'isv6' is B_FALSE)
7420   */

7422  ifr = (struct ifreq *)mp1->b_rptr;

7424  rw_enter(&ipst->ips_ill_g_lock, RW_READER);
7425  ill = ILL_START_WALK_V4(&ctx, ipst);
7426  for (; ill != NULL; ill = ill_next(&ctx, ill)) {
7427      if (IS_UNDER_IPMP(ill))
7428          continue;
7429      for (ipif = ill->ill_ipif; ipif != NULL;
7430          ipif = ipif->ipif_next) {
7431          if (zoneid != ipif->ipif_zoneid &&
7432              ipif->ipif_zoneid != ALL_ZONES)
7433              continue;
7434          if ((uchar_t *)&ifr[1] > mp1->b_wptr) {
7435              if (iocp->ioc_cmd == O_SIOCGIFCONF) {
7436                  /* old behaviour */
7437                  rw_exit(&ipst->ips_ill_g_lock);
7438                  return (EINVAL);
7439              } else {
7440                  goto if_copydone;
7441              }
7442          }
7443          ipif_get_name(ipif, ifr->ifr_name,
7444              sizeof (ifr->ifr_name));
7445          sin = (sin_t *)&ifr->ifr_addr;
7446          *sin = sin_null;
7447          sin->sin_family = AF_INET;
7448          sin->sin_addr.s_addr = ipif->ipif_lcl_addr;
7449          ifr++;
7450      }
7451  }
7452  if_copydone:
7453  rw_exit(&ipst->ips_ill_g_lock);

```

```

7454      mp1->b_wptr = (uchar_t *)ifr;

7456      if (STRUCT_BUF(ifc) != NULL) {
7457          STRUCT_FSET(ifc, ifc_len,
7458              (int)((uchar_t *)ifr - mp1->b_rptr));
7459      }
7460      return (0);
7461  }

7463  /*
7464   * Get the interfaces using the address hosted on the interface passed in,
7465   * as a source address
7466   */
7467  /* ARGSUSED */
7468  int
7469  ip_siocctl_get_lifsrcof(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q,
7470      mblk_t *mp, ip_ioctl_cmd_t *ipip, void *ifreq)
7471  {
7472      mblk_t *mp1;
7473      ill_t *ill, *ill_head;
7474      ipif_t *ipif, *orig_ipif;
7475      int numlifs = 0;
7476      size_t lifs_bufsize, lifsmaxlen;
7477      struct lifreq *lifr;
7478      struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
7479      uint_t ifindex;
7480      zoneid_t zoneid;
7481      boolean_t isv6 = B_FALSE;
7482      struct sockaddr_in *sin;
7483      struct sockaddr_in6 *sin6;
7484      STRUCT_HANDLE(lifsrcof, lifs);
7485      ip_stack_t *ipst;

7487      ipst = CONNQ_TO_IPST(q);

7489      ASSERT(q->q_next == NULL);

7491      zoneid = Q_TO_CONN(q)->conn_zoneid;

7493      /* Existence verified in ip_wput_nondata */
7494      mp1 = mp->b_cont->b_cont;

7496      /*
7497       * Must be (better be!) continuation of a TRANSPARENT
7498       * IOCTL. We just copied in the lifsrcof structure.
7499       */
7500      STRUCT_SET_HANDLE(lifs, iocp->ioc_flag,
7501          (struct lifsrcof *)mp1->b_rptr);

7503      if (MBLKL(mp1) != STRUCT_SIZE(lifs))
7504          return (EINVAL);

7506      ifindex = STRUCT_FGET(lifs, lifs_ifindex);
7507      isv6 = (Q_TO_CONN(q)->conn_family == AF_INET6;
7508      ipif = ipif_lookup_on_ifindex(ifindex, isv6, zoneid, ipst);
7509      if (ipif == NULL) {
7510          ipldbg(("ip_siocctl_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_usesrc_lock protects ill_usesrc_grp_next */
7544     rw_enter(&ipst->ips_ill_g_usesrc_lock, RW_READER);
7545     rw_enter(&ipst->ips_ill_g_lock, RW_READER);

7547     ill = ill->ill_usesrc_grp_next; /* start from next ill */
7548     for (; (ill != NULL) && (ill != ill_head);
7549         ill = ill->ill_usesrc_grp_next) {

7551         if ((uchar_t *)&lifr[1] > 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_usesrc_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 *ipip, 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 lifclen;
7601     zoneid_t zoneid;
7602     STRUCT_HANDLE(lifconf, lifc);
7603     ip_stack_t *ipst = CONNQ_TO_IPST(q);

7605     ipldbg(("ip_ioctl_get_lifconf"));

7607     ASSERT(q->q_next == NULL);

7609     zoneid = Q_TO_CONN(q)->conn_zoneid;

7611     /* Existence verified in ip_wput_nondata */
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_FGET(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_siocctl_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     greply(q, mp);
7867 }

7869 static void
7870 ip_siocctl_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;

7922         /*
7923          * ip_addr_scope_v6() and ip6_asp_lookup() handle
7924          * v4 mapped addresses; ire_ftable_lookup_v6()
7925          * and ip_select_source_v6() do not.
7926          */
7927         dir->dir_dscope = ip_addr_scope_v6(daddr);
7928         dlabel = ip6_asp_lookup(daddr, &dir->dir_precedence, ipst);

7930         isipv4 = IN6_IS_ADDR_V4MAPPED(daddr);
7931         if (isipv4) {
7932             IN6_V4MAPPED_TO_IPADDR(daddr, v4daddr);
7933             v4setsrc = INADDR_ANY;
7934             ire = ire_route_recursive_v4(v4daddr, 0, NULL, zoneid,
7935             NULL, match_ire, IRR_ALLOCATE, 0, ipst, &v4setsrc,
7936             NULL, NULL);
7937         } else {
7938             v6setsrc = ipv6_all_zeros;
7939             ire = ire_route_recursive_v6(daddr, 0, NULL, zoneid,
7940             NULL, match_ire, IRR_ALLOCATE, 0, ipst, &v6setsrc,
7941             NULL, NULL);
7942         }
7943         ASSERT(ire != NULL);
7944         if (ire->ire_flags & (RTF_REJECT|RTF_BLACKHOLE)) {
7945             ire_refrele(ire);
7946             dir->dir_dreachable = 0;

7948             /* move on to next dst addr */
7949             continue;
7950         }
7951         dir->dir_dreachable = 1;

7953         dst_ill = ire_nexthop_ill(ire);
7954         if (dst_ill == NULL) {
7955             ire_refrele(ire);
7956             continue;
7957         }

7959         /* With ipmp we most likely look at the ipmp ill here */
7960         dir->dir_dmactype = dst_ill->ill_mactype;

7962         if (isipv4) {
7963             ipaddr_t v4saddr;

7965             if (ip_select_source_v4(dst_ill, v4setsrc, v4daddr,
7966             connp->conn_ixa->ixa_multicast_ifaddr, zoneid, ipst,
7967             &v4saddr, NULL, &ipif_flags) != 0) {
7968                 v4saddr = INADDR_ANY;
7969                 ipif_flags = 0;
7970             }
7971             IN6_IPADDR_TO_V4MAPPED(v4saddr, saddr);
7972         } else {
7973             if (ip_select_source_v6(dst_ill, &v6setsrc, daddr,
7974             zoneid, ipst, B_FALSE, IPV6_PREFER_SRC_DEFAULT,
7975             saddr, NULL, &ipif_flags) != 0) {
7976                 *saddr = ipv6_all_zeros;
7977                 ipif_flags = 0;
7978             }
7979         }

7981         dir->dir_sscope = ip_addr_scope_v6(saddr);

```

```

7982         slabel = ip6_asp_lookup(saddr, NULL, ipst);
7983         dir->dir_labelmatch = ip6_asp_labelcmp(dlabel, slabel);
7984         dir->dir_sdeprecated = (ipif_flags & IPIF_DEPRECATED) ? 1 : 0;
7985         ire_refrele(ire);
7986         ill_refrele(dst_ill);
7987     }
7988     miocack(q, mp, iocp->ioc_count, 0);
7989 }

7991 /*
7992  * Check if this is an address assigned to this machine.
7993  * Skips interfaces that are down by using ire checks.
7994  * Translates mapped addresses to v4 addresses and then
7995  * treats them as such, returning true if the v4 address
7996  * associated with this mapped address is configured.
7997  * Note: Applications will have to be careful what they do
7998  * with the response; use of mapped addresses limits
7999  * what can be done with the socket, especially with
8000  * respect to socket options and ioctls - neither IPv4
8001  * options nor IPv6 sticky options/ancillary data options
8002  * may be used.
8003  */
8004 /* ARGSUSED */
8005 int
8006 ip_siocctl_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;

8016     ipldbg(("ip_siocctl_tmyaddr"));

8018     ASSERT(q->q_next == NULL); /* this ioctl not allowed if ip is module */
8019     zoneid = Q_TO_CONN(q)->conn_zoneid;
8020     ipst = CONNQ_TO_IPST(q);

8022     /* Existence verified in ip_wput_nondata */
8023     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;

8030         if (IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
8031             ipaddr_t v4_addr;

8033             IN6_V4MAPPED_TO_IPADDR(&sin6->sin6_addr,
8034             v4_addr);
8035             ire = ire_ftable_lookup_v4(v4_addr, 0, 0,
8036             IRE_LOCAL|IRE_LOOPBACK, NULL, zoneid, NULL,
8037             MATCH_IRE_TYPE | MATCH_IRE_ZONEONLY, 0, ipst, NULL);
8038         } else {
8039             in6_addr_t v6addr;

8041             v6addr = sin6->sin6_addr;
8042             ire = ire_ftable_lookup_v6(&v6addr, 0, 0,
8043             IRE_LOCAL|IRE_LOOPBACK, NULL, zoneid, NULL,
8044             MATCH_IRE_TYPE | MATCH_IRE_ZONEONLY, 0, ipst, NULL);
8045         }
8046         break;
8047     }

```

```

8048     case AF_INET: {
8049         ipaddr_t v4addr;

8051         v4addr = sin->sin_addr.s_addr;
8052         ire = ire_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 *ipip, 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     ipldbg(("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 *ipip, 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 *ipip, void *dummy_ifreq)
8177 {
8178     int err;
8179     ipaddr_t ipaddr;

```

```

8180 struct iocblk *iocp;
8181 conn_t *connp;
8182 struct arpreq *ar;
8183 struct xarpreq *xar;
8184 int arp_flags, flags, alength;
8185 uchar_t lladdr;
8186 ip_stack_t *ipst;
8187 ill_t *ill = ipif->ipif_ill;
8188 ill_t *proxy_ill = NULL;
8189 ipmp_arpent_t *entp = NULL;
8190 boolean_t proxyarp = B_FALSE;
8191 boolean_t if_arp_ioctl = B_FALSE;
8192 ncec_t *ncec = NULL;
8193 nce_t *nce;

8195 ASSERT(!(q->q_flag & QREADR) && q->q_next == NULL);
8196 connp = Q_TO_CONN(q);
8197 ipst = connp->conn_netstack->netstack_ip;
8198 iocp = (struct iocblk *)mp->b_rptr;

8200 if (ipip->ipi_cmd_type == XARP_CMD) {
8201     /* We have a chain - M_IOCTL-->MI_COPY_MBLK-->XARPREQ_MBLK */
8202     xar = (struct xarpreq *)mp->b_cont->b_cont->b_rptr;
8203     ar = NULL;

8205     arp_flags = xar->xarp_flags;
8206     lladdr = (uchar_t *)LLADDR(&xar->xarp_ha);
8207     if_arp_ioctl = (xar->xarp_ha.sdl_nlen != 0);
8208     /*
8209      * Validate against user's link layer address length
8210      * input and name and addr length limits.
8211      */
8212     alength = ill->ill_phys_addr_length;
8213     if (ipip->ipi_cmd == SIOCSXARP) {
8214         if (alength != xar->xarp_ha.sdl_alen ||
8215             (alength + xar->xarp_ha.sdl_nlen >
8216              sizeof(xar->xarp_ha.sdl_data)))
8217             return (EINVAL);
8218     }
8219 } else {
8220     /* We have a chain - M_IOCTL-->MI_COPY_MBLK-->ARPREQ_MBLK */
8221     ar = (struct arpreq *)mp->b_cont->b_cont->b_rptr;
8222     xar = NULL;

8224     arp_flags = ar->arp_flags;
8225     lladdr = (uchar_t *)ar->arp_ha.sa_data;
8226     /*
8227      * Theoretically, the sa_family could tell us what link
8228      * layer type this operation is trying to deal with. By
8229      * common usage AF_UNSPEC means ethernet. We'll assume
8230      * any attempt to use the SIOC*ARP ioctls is for ethernet,
8231      * for now. Our new SIOC*XARP ioctls can be used more
8232      * generally.
8233      *
8234      * If the underlying media happens to have a non 6 byte
8235      * address, arp module will fail set/get, but the del
8236      * operation will succeed.
8237      */
8238     alength = 6;
8239     if ((ipip->ipi_cmd != SIOCDAARP) &&
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->nccommon;

8298 switch (iocp->ioc_cmd) {
8299 case SIOCDAARP:
8300 case SIOCDAARP: {
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_siocctl_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 (ncec != 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 (ncec != 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 refhold and return the associated ipif via 'ci'.
8413 */
8414 int
8415 ip_extract_arpreq(queue_t *q, mblk_t *mp, const ip_ioctl_cmd_t *ipip,
8416     cmd_info_t *ci)
8417 {
8418     mblk_t *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 (ipip->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_rptr;
8455         sin = (sin_t *)&ar->arp_pa;
8456     }

8458     if (ipip->ipi_cmd_type == XARP_CMD && sdl->sdl_nlen != 0) {
8459         ipif = ipif_lookup_on_name(sdl->sdl_data, sdl->sdl_nlen,
8460             B_FALSE, &exists, B_FALSE, ALL_ZONES, ipst);
8461         if (ipif == NULL)
8462             return (ENXIO);
8463         if (ipif->ipif_id != 0) {
8464             ipif_refrele(ipif);
8465             return (ENXIO);
8466         }
8467     } else {
8468         /*
8469          * Either an SIOC[DGS]ARP or an SIOC[DGS]XARP with an sdl_nlen
8470          * of 0: use the IP address to find the ipif. If the IP
8471          * address is an IPMP test address, ire_ftable_lookup() will
8472          * find the wrong ill, so we first do an ipif_lookup_addr().
8473          */
8474         ipif = ipif_lookup_addr(sin->sin_addr.s_addr, NULL, ALL_ZONES,
8475             ipst);
8476         if (ipif == NULL) {
8477             ire = ire_ftable_lookup_v4(sin->sin_addr.s_addr,
8478                 0, 0, IRE_IF_RESOLVER, NULL, ALL_ZONES,
8479                 NULL, MATCH_IRE_TYPE, 0, ipst, NULL);
8480             if (ire == NULL || ((ill = ire->ire_ill) == NULL)) {
8481                 if (ire != NULL)
8482                     ire_refrele(ire);
8483                 return (ENXIO);
8484             }
8485             ASSERT(ire != NULL && ill != NULL);
8486             ipif = ill->ill_ipif;
8487             ipif_refhold(ipif);
8488             ire_refrele(ire);
8489         }
8490     }

8492     if (ipif->ipif_ill->ill_net_type != IRE_IF_RESOLVER) {
8493         ipif_refrele(ipif);
8494         return (ENXIO);
8495     }

8497     ci->ci_sin = sin;
8498     ci->ci_ipif = ipif;
8499     return (0);
8500 }

8502 /*
8503  * Link or unlink the illgrp on IPMP meta-interface 'ill' depending on the
8504  * value of 'ioccmd'. While an illgrp is linked to an ipmp_grp_t, it is
8505  * accessible from that ipmp_grp_t, which means 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_ioctl_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_ioctl_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     }
8844     return (B_TRUE);
8845 }
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->bp;
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;
8871     ASSERT(q->q_next == NULL);
8872     ASSERT(CONN_Q(q));
8874     if (!getset_ioctl_checks(mp)) {
8875         miocnak(q, mp, 0, EINVAL);
8876         return;
8877     }
8878     ipst = CONNQ_TO_IPST(q);
8879     stack = ipst->ips_netstack;
8880     pioc = (mod_ioc_prop_t *)mpl->b_rptr;
8882     switch (pioc->mpr_proto) {
8883     case MOD_PROTO_IP:
8884     case MOD_PROTO_IPV4:
8885     case MOD_PROTO_IPV6:
8886         ptbl = ipst->ips_propinfo_tbl;
8887         cbarg = ipst;
8888         break;
8889     case MOD_PROTO_RAWIP:
8890         is = stack->netstack_icmp;
8891         ptbl = is->is_propinfo_tbl;
8892         cbarg = is;
8893         break;
8894     case MOD_PROTO_TCP:
8895         tcps = stack->netstack_tcp;
8896         ptbl = tcps->tcps_propinfo_tbl;
8897         cbarg = tcps;
8898         break;
8899     case MOD_PROTO_UDP:
8900         us = stack->netstack_udp;
8901         ptbl = us->us_propinfo_tbl;
8902         cbarg = us;
8903         break;
8904     case MOD_PROTO_SCTP:
8905         sctps = stack->netstack_sctp;

```

```

8906         ptbl = sctps->sctps_propinfo_tbl;
8907         cbarg = sctps;
8908         break;
8909     case MOD_PROTO_DCCP:
8910         dccps = stack->netstack_dccp;
8911         ptbl = dccps->dccps_propinfo_tbl;
8912         cbarg = dccps;
8913 #endif /* ! codereview */
8914     default:
8915         miocnak(q, mp, 0, EINVAL);
8916         return;
8917     }
8919     /* search for given property in respective protocol propinfo table */
8920     for (pinfo = ptbl; pinfo->mpi_name != NULL; pinfo++) {
8921         if (strcmp(pinfo->mpi_name, pioc->mpr_name) == 0 &&
8922             pinfo->mpi_proto == pioc->mpr_proto)
8923             break;
8924     }
8925     if (pinfo->mpi_name == NULL) {
8926         miocnak(q, mp, 0, ENOENT);
8927         return;
8928     }
8930     set = (iocp->ioc_cmd == SIOCSETPROP) ? B_TRUE : B_FALSE;
8931     if (set && pinfo->mpi_setf != NULL) {
8932         cr = msg_getcred(mp, NULL);
8933         if (cr == NULL)
8934             cr = iocp->ioc_cr;
8935         err = pinfo->mpi_setf(cbarg, cr, pinfo, pioc->mpr_ifname,
8936             pioc->mpr_val, pioc->mpr_flags);
8937     } else if (!set && pinfo->mpi_getf != NULL) {
8938         err = pinfo->mpi_getf(cbarg, pinfo, pioc->mpr_ifname,
8939             pioc->mpr_val, pioc->mpr_valsize, pioc->mpr_flags);
8940     } else {
8941         err = EPERM;
8942     }
8944     if (err != 0) {
8945         miocnak(q, mp, 0, err);
8946     } else {
8947         if (set)
8948             miocack(q, mp, 0, 0);
8949         else /* For get, we need to return back the data */
8950             miocack(q, mp, iocp->ioc_count, 0);
8951     }
8952 }
8954 /*
8955  * process the legacy ND_GET, ND_SET ioctl just for {ip|ip6}_forwarding
8956  * as several routing daemons have unfortunately used this 'unpublished'
8957  * but well-known ioctls.
8958  */
8959 /* ARGSUSED */
8960 static void
8961 ip_process_legacy_nddprop(queue_t *q, mblk_t *mp)
8962 {
8963     struct iocblk *iocp = (struct iocblk *)mp->bp;
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;
8971     ASSERT(CONN_Q(q));

```

```

8972     ipst = CONNQ_TO_IPST(q);

8974     if (iocp->ioc_count == 0 || mp1 == NULL) {
8975         miocnak(q, mp, 0, EINVAL);
8976         return;
8977     }

8979     mp1->b_datap->db_lim[-1] = '\0';          /* Force null termination */
8980     pval = buf = pname = (char *)mp1->b_rptr;
8981     bufsize = MBLKL(mp1);

8983     if (strcmp(pname, "ip_forwarding") == 0) {
8984         pname = "forwarding";
8985         proto = MOD_PROTO_IPV4;
8986     } else if (strcmp(pname, "ip6_forwarding") == 0) {
8987         pname = "forwarding";
8988         proto = MOD_PROTO_IPV6;
8989     } else {
8990         miocnak(q, mp, 0, EINVAL);
8991         return;
8992     }

8994     ptbl = ipst->ips_propinfo_tbl;
8995     for (pinfo = ptbl; pinfo->mpi_name != NULL; pinfo++) {
8996         if (strcmp(pinfo->mpi_name, pname) == 0 &&
8997             pinfo->mpi_proto == proto)
8998             break;
8999     }

9001     ASSERT(pinfo->mpi_name != NULL);

9003     switch (iocp->ioc_cmd) {
9004     case ND_GET:
9005         if ((err = pinfo->mpi_getf(ipst, pinfo, NULL, buf, bufsize,
9006             0)) == 0) {
9007             miocack(q, mp, iocp->ioc_count, 0);
9008             return;
9009         }
9010         break;
9011     case ND_SET:
9012         /*
9013          * buffer will have property name and value in the following
9014          * format,
9015          * <property name>'\0'<property value>'\0', extract them;
9016          */
9017         while (*pval++)
9018             noop;

9020         if (!*pval || pval >= (char *)mp1->b_wptr) {
9021             err = EINVAL;
9022         } else if ((err = pinfo->mpi_setf(ipst, NULL, pinfo, NULL,
9023             pval, 0)) == 0) {
9024             miocack(q, mp, 0, 0);
9025             return;
9026         }
9027         break;
9028     default:
9029         err = EINVAL;
9030         break;
9031     }
9032     miocnak(q, mp, 0, err);
9033 }

9035 /*
9036  * Wrapper function for resuming deferred ioctl processing
9037  * Used for SIOCGDSTINFO, SIOCGIP6ADDRPOLICY, SIOCGMSFILTER,

```

```

9038  * SIOCSMSFILTER, SIOCGIPMSFILTER, and SIOCSIPMSFILTER currently.
9039  */
9040  /* ARGSUSED */
9041  void
9042  ip_siocctl_copyin_resume(ipsq_t *dummy_ipsq, queue_t *q, mblk_t *mp,
9043      void *dummy_arg)
9044  {
9045      ip_siocctl_copyin_setup(q, mp);
9046  }

9048  /*
9049  * ip_siocctl_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_siocctl_copyin_setup(queue_t *q, mblk_t *mp)
9058  {
9059      int copyin_size;
9060      struct iocblk *iocp = (struct iocblk *)mp->b_rptr;
9061      ip_ioctl_cmd_t *ipip;
9062      cred_t *cr;
9063      ip_stack_t *ipst;

9065      if (CONNQ(q))
9066          ipst = CONNQ_TO_IPST(q);
9067      else
9068          ipst = ILLQ_TO_IPST(q);

9070      ipip = ip_siocctl_lookup(iocp->ioc_cmd);
9071      if (ipip == NULL) {
9072          /*
9073           * The ioctl is not one we understand or own.
9074           * Pass it along to be processed down stream,
9075           * if this is a module instance of IP, else nak
9076           * the ioctl.
9077           */
9078          if (q->q_next == NULL) {
9079              goto nak;
9080          } else {
9081              putnext(q, mp);
9082              return;
9083          }
9084      }

9086      /*
9087       * If this is deferred, then we will do all the checks when we
9088       * come back.
9089       */
9090      if ((iocp->ioc_cmd == SIOCGDSTINFO ||
9091          iocp->ioc_cmd == SIOCGIP6ADDRPOLICY) && !ip6_asp_can_lookup(ipst)) {
9092          ip6_asp_pending_op(q, mp, ip_siocctl_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 crfree()) by
9117      * the framework; the caller of ioctl needs to hold the reference
9118      * for the duration of the call).
9119      */
9120     cr = msg_getcred(mp, NULL);
9121     if (cr == NULL)
9122         cr = iocp->ioc_cr;

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 SIOCGLIFSRCOF:
9165         copyin_size = SIZEOF_STRUCT(lifsrcof, 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     ipldbg("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(CONN_Q(q));
9350     connp = Q_TO_CONN(q);
9351     isv6 = (connp->conn_family == AF_INET6);
9352     zoneid = connp->conn_zoneid;
9353     namelen = mi_strlen(name);
9354     if (namelen == 0)
9355         return (EINVAL);

9357     exists = B_FALSE;
9358     if ((namelen + 1 == sizeof(ipif_loopback_name)) &&
9359         (mi_strcmp(name, ipif_loopback_name) == 0)) {
9360         /*
9361          * Allow creating lo0 using SIOCLIFADDIF.
9362          * can't be any other writer thread. So can pass null below
9363          * for the last 4 args to ipif_lookup_name.
9364          */
9365         ipif = ipif_lookup_on_name(lifr->lifr_name, namelen, B_TRUE,
9366             &exists, isv6, zoneid, ipst);
9367         /* Prevent any further action */

```

```

9368     if (ipif == NULL) {
9369         return (ENOBUFS);
9370     } else if (!exists) {
9371         /* We created the ipif now and as writer */
9372         ipif_refrele(ipif);
9373         return (0);
9374     } else {
9375         ill = ipif->ipif_ill;
9376         ill_refhold(ill);
9377         ipif_refrele(ipif);
9378     }
9379 } else {
9380     /* Look for a colon in the name. */
9381     endp = &name[namelen];
9382     for (cp = endp; --cp > name; ) {
9383         if (*cp == IPIF_SEPARATOR_CHAR) {
9384             found_sep = B_TRUE;
9385             /*
9386              * Reject any non-decimal aliases for plumbing
9387              * of logical interfaces. Aliases with leading
9388              * zeroes are also rejected as they introduce
9389              * ambiguity in the naming of the interfaces.
9390              * Comparing with "0" takes care of all such
9391              * cases.
9392              */
9393             if ((strncmp("0", cp+1, 1)) == 0)
9394                 return (EINVAL);
9395
9396             if (ddi_strotol(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%c%d", ill->ill_name,
9449     IPIF_SEPARATOR_CHAR, ipif->ipif_id);
9450 ipldbg(("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 *pip, 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     ipldbg(("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 *pip, 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("!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 bcst ire's 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) untimout(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

```

```

9986      * in this address getting automatically reconfigured from under the
9987      * administrator.
9988      */
9989      if (ipif->ipif_isv6 && ipif->ipif_id == 0) {
9990          if (iocp == NULL || (iocp->ioc_cmd == SIOCSLIFADDR &&
9991              !IN6_IS_ADDR_UNSPECIFIED(&v6addr)))
9992              ill->ill_manual_linklocal = 1;
9993      }

9994      /*
9995       * When publishing an interface address change event, we only notify
9996       * the event listeners of the new address. It is assumed that if they
9997       * actively care about the addresses assigned that they will have
9998       * already discovered the previous address assigned (if there was one.)
9999       *
10000      * Don't attach nic event message for SIOCLIFADDIF ioctl.
10001      */
10002      if (iocp != NULL && iocp->ioc_cmd != SIOCLIFADDIF) {
10003          ill_nic_event_dispatch(ill, MAP_IPIF_ID(ipif->ipif_id),
10004              NE_ADDRESS_CHANGE, sin, sinlen);
10005      }

10006      mutex_exit(&ill->ill_lock);

10007      if (need_up) {
10008          /*
10009           * Now bring the interface back up. If this
10010           * is the only IPIF for the ILL, ipif_up
10011           * will have to re-bind to the device, so
10012           * we may get back EINPROGRESS, in which
10013           * case, this IOCTL will get completed in
10014           * ip_rput_dlpi when we see the DL_BIND_ACK.
10015           */
10016          err = ipif_up(ipif, q, mp);
10017      } else {
10018          /* Perhaps ilgs should use this ill */
10019          update_conn_ill(NULL, ill->ill_ipst);
10020      }

10021      if (need_dl_down)
10022          ill_dl_down(ill);

10023      if (need_arp_down && !ill->ill_isv6)
10024          (void) ipif_arp_down(ipif);

10025      /*
10026       * The default multicast interface might have changed (for
10027       * instance if the IPv6 scope of the address changed)
10028       */
10029      ire_increment_multicast_generation(ill->ill_ipst, ill->ill_isv6);

10030      return (err);
10031 }

10032 /*
10033  * Restart entry point to restart the address set operation after the
10034  * refcounts have dropped to zero.
10035  */
10036 /* ARGSUSED */
10037 int
10038 ip_ioctl_addr_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10039     ip_ioctl_cmd_t *ipip, void *ifreq)
10040 {
10041     ipldbg(("ip_ioctl_addr_restart(%s:%u %p)\n",
10042         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10043     ASSERT(IAM_WRITER_IPIF(ipif));

```

```

10044     (void) ipif_down_tail(ipif);
10045     return (ip_ioctl_addr_tail(ipif, sin, q, mp, B_TRUE));
10046 }

10047 /* ARGSUSED */
10048 int
10049 ip_ioctl_get_addr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10050     ip_ioctl_cmd_t *ipip, void *if_req)
10051 {
10052     sin6_t *sin6 = (struct sockaddr_in6 *)sin;
10053     struct lifreq *lifr = (struct lifreq *)if_req;

10054     ipldbg(("ip_ioctl_get_addr(%s:%u %p)\n",
10055         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10056     /*
10057      * The net mask and address can't change since we have a
10058      * reference to the ipif. So no lock is necessary.
10059      */
10060     if (ipif->ipif_isv6) {
10061         *sin6 = sin6_null;
10062         sin6->sin6_family = AF_INET6;
10063         sin6->sin6_addr = ipif->ipif_v6lcl_addr;
10064         ASSERT(ipip->ipi_cmd_type == LIF_CMD);
10065         lifr->lifr_addrlen =
10066             ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
10067     } else {
10068         *sin = sin_null;
10069         sin->sin_family = AF_INET;
10070         sin->sin_addr.s_addr = ipif->ipif_lcl_addr;
10071         if (ipip->ipi_cmd_type == LIF_CMD) {
10072             lifr->lifr_addrlen =
10073                 ip_mask_to_plen(ipif->ipif_net_mask);
10074         }
10075     }
10076     return (0);
10077 }

10078 /*
10079  * Set the destination address for a pt-pt interface.
10080  */
10081 /* ARGSUSED */
10082 int
10083 ip_ioctl_dstaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10084     ip_ioctl_cmd_t *ipip, void *if_req)
10085 {
10086     int err = 0;
10087     in6_addr_t v6addr;
10088     boolean_t need_up = B_FALSE;

10089     ipldbg(("ip_ioctl_dstaddr(%s:%u %p)\n",
10090         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10091     ASSERT(IAM_WRITER_IPIF(ipif));

10092     if (ipif->ipif_isv6) {
10093         sin6_t *sin6;

10094         if (sin->sin_family != AF_INET6)
10095             return (EAFNOSUPPORT);

10096         sin6 = (sin6_t *)sin;
10097         v6addr = sin6->sin6_addr;

10098         if (!ip_remote_addr_ok_v6(&v6addr, &ipif->ipif_v6net_mask))
10099             return (EADDRNOTAVAIL);
10100     } else {
10101         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_siocctl_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_siocctl_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_siocctl_dstaddr_tail(ipif, sin, q, mp, B_TRUE));
10175 }
10177 /* ARGSUSED */
10178 int
10179 ip_siocctl_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_siocctl_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_siocctl_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 SIOCSLIFFLAGS 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_siocctl_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_siocctl_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_PHYINT_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_v6cl_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 (!ipif->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_siocctl_flags_restart(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10721                          ip_ioctl_cmd_t *ipip, void *if_req)
10722  {
10723      uint64_t flags;
10724      struct ifreq *ifr = if_req;
10725      struct lifreq *lifr = if_req;
10726      uint64_t turn_on, turn_off;

10728      ipldbg(("ip_siocctl_flags_restart(%s:%u %p)\n",
10729              ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));

10731      if (ipip->ipi_cmd_type == IF_CMD) {
10732          /* cast to uint16_t prevents unwanted sign extension */
10733          flags = (uint16_t)ifr->ifr_flags;
10734      } else {
10735          flags = lifr->lifr_flags;
10736      }

10738      /*
10739       * If this function call is a result of the ILLF_NOACCEPT flag
10740       * change, do not call ipif_down_tail(). See ip_siocctl_flags().
10741       */
10742      ip_siocctl_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_siocctl_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_siocctl_get_flags(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10755                      ip_ioctl_cmd_t *ipip, void *if_req)
10756  {
10757      /*
10758       * Has the flags been set correctly till now ?
10759       */
10760      ill_t *ill = ipif->ipif_ill;
10761      phyint_t *phyi = ill->ill_phyint;

10763      ipldbg(("ip_siocctl_get_flags(%s:%u %p)\n",
10764              ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
10765      ASSERT((phyi->phyint_flags & ~(IFF_PHYINT_FLAGS)) == 0);
10766      ASSERT((ill->ill_flags & ~(IFF_PHYINTINST_FLAGS)) == 0);
10767      ASSERT((ipif->ipif_flags & ~(IFF_LOGINT_FLAGS)) == 0);

10769      /*
10770       * Need a lock since some flags can be set even when there are
10771       * references to the ipif.
10772       */
10773      mutex_enter(&ill->ill_lock);
10774      if (ipip->ipi_cmd_type == IF_CMD) {
10775          struct ifreq *ifr = (struct ifreq *)if_req;

10777          /* Get interface flags (low 16 only). */
10778          ifr->ifr_flags = ((ipif->ipif_flags |
10779                          ill->ill_flags | phyi->phyint_flags) & 0xffff);
10780      } else {
10781          struct lifreq *lifr = (struct lifreq *)if_req;

10783          /* Get interface flags. */
10784          lifr->lifr_flags = ipif->ipif_flags |
10785                          ill->ill_flags | phyi->phyint_flags;
10786      }
10787      mutex_exit(&ill->ill_lock);
10788      return (0);
10789  }

10791  /*
10792   * We allow the MTU to be set on an ILL, but not have it be different
10793   * for different IPIFs since we don't actually send packets on IPIFs.
10794   */
10795  /* ARGSUSED */
10796  int
10797  ip_siocctl_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10798                ip_ioctl_cmd_t *ipip, void *if_req)
10799  {
10800      int mtu;
10801      int ip_min_mtu;
10802      struct ifreq *ifr;
10803      struct lifreq *lifr;
10804      ill_t *ill;

10806      ipldbg(("ip_siocctl_mtu(%s:%u %p)\n", ipif->ipif_ill->ill_name,
10807              ipif->ipif_id, (void *)ipif));
10808      if (ipip->ipi_cmd_type == IF_CMD) {
10809          ifr = (struct ifreq *)if_req;
10810          mtu = ifr->ifr_metric;
10811      } else {
10812          lifr = (struct lifreq *)if_req;
10813          mtu = lifr->lifr_mtu;
10814      }
10815      /* Only allow for logical unit zero i.e. not on "bge0:17" */
10816      if (ipif->ipif_id != 0)
10817          return (EINVAL);

10819      ill = ipif->ipif_ill;

```



```

10820     if (ipif->ipif_isv6)
10821         ip_min_mtu = IPV6_MIN_MTU;
10822     else
10823         ip_min_mtu = IP_MIN_MTU;

10825     mutex_enter(&ill->ill_lock);
10826     if (mtu > ill->ill_max_frag || mtu < ip_min_mtu) {
10827         mutex_exit(&ill->ill_lock);
10828         return (EINVAL);
10829     }
10830     /* Avoid increasing ill_mc_mtu */
10831     if (ill->ill_mc_mtu > mtu)
10832         ill->ill_mc_mtu = mtu;

10834     /*
10835      * The dce and fragmentation code can handle changes to ill_mtu
10836      * concurrent with sending/fragmenting packets.
10837      */
10838     ill->ill_mtu = mtu;
10839     ill->ill_flags |= ILLF_FIXEDMTU;
10840     mutex_exit(&ill->ill_lock);

10842     /*
10843      * Make sure all dce_generation checks find out
10844      * that ill_mtu/ill_mc_mtu has changed.
10845      */
10846     dce_increment_all_generations(ill->ill_isv6, ill->ill_ipst);

10848     /*
10849      * Refresh IPMP meta-interface MTU if necessary.
10850      */
10851     if (IS_UNDER_IPMP(ill))
10852         ipmp_illgrp_refresh_mtu(ill->ill_grp);

10854     /* Update the MTU in SCTP's list */
10855     sctp_update_ipif(ipif, SCTP_IPIF_UPDATE);
10856     return (0);
10857 }

10859 /* Get interface MTU. */
10860 /* ARGSUSED */
10861 int
10862 ip_ioctl_get_mtu(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10863     ip_ioctl_cmd_t *ipip, void *if_req)
10864 {
10865     struct ifreq    *ifr;
10866     struct lifreq    *lifr;

10868     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 (ipip->ipi_cmd_type == IF_CMD) {
10876         ifr = (struct ifreq *)if_req;
10877         ifr->ifr_metric = ipif->ipif_ill->ill_mtu;
10878     } else {
10879         lifr = (struct lifreq *)if_req;
10880         lifr->lifr_mtu = ipif->ipif_ill->ill_mtu;
10881     }
10882     return (0);
10883 }

10885 /* Set interface broadcast address. */

```

```

10886 /* ARGSUSED2 */
10887 int
10888 ip_ioctl_brdaddr(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
10889     ip_ioctl_cmd_t *ipip, void *if_req)
10890 {
10891     ipaddr_t addr;
10892     ire_t    *ire;
10893     ill_t     *ill = ipif->ipif_ill;
10894     ip_stack_t *ipst = ill->ill_ipst;

10896     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 *ipip, 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 *ipip, 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 *ipip, 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 *ipip, void *if_req)
11077 {
11078     struct lifreq *lifr = (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_addrlen =
11093          ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
11094  } else {
11095      *sin = sin_null;
11096      sin->sin_family = AF_INET;
11097      sin->sin_addr.s_addr = ipif->ipif_net_mask;
11098      if (ipip->ipi_cmd_type == LIF_CMD) {
11099          lifr->lifr_addrlen =
11100              ip_mask_to_plen(ipif->ipif_net_mask);
11101      }
11102  }
11103  return (0);
11104 }

11106 /* ARGSUSED */
11107 int
11108 ip_siocctl_metric(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11109 ip_ioctl_cmd_t *pip, void *if_req)
11110 {
11111     ipldbg(("ip_siocctl_metric(%s:%u %p)\n",
11112         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11113
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);
11120
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;
11128
11129         ifr = (struct ifreq *)if_req;
11130         ipif->ipif_ill->ill_metric = ifr->ifr_metric;
11131     } else {
11132         struct lifreq *lifr;
11133
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_siocctl_get_metric(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11143 ip_ioctl_cmd_t *pip, void *if_req)
11144 {
11145     /* Get interface metric. */
11146     ipldbg(("ip_siocctl_get_metric(%s:%u %p)\n",
11147         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11148
11149     if (ipip->ipi_cmd_type == IF_CMD) {

```

```

11150         struct ifreq *ifr;
11151
11152         ifr = (struct ifreq *)if_req;
11153         ifr->ifr_metric = ipif->ipif_ill->ill_metric;
11154     } else {
11155         struct lifreq *lifr;
11156
11157         lifr = (struct lifreq *)if_req;
11158         lifr->lifr_metric = ipif->ipif_ill->ill_metric;
11159     }
11160
11161     return (0);
11162 }

11164 /* ARGSUSED */
11165 int
11166 ip_siocctl_muxid(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11167 ip_ioctl_cmd_t *pip, void *if_req)
11168 {
11169     int arp_muxid;
11170
11171     ipldbg(("ip_siocctl_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;
11178
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;
11183
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_siocctl_get_muxid(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11194 ip_ioctl_cmd_t *pip, void *if_req)
11195 {
11196     int arp_muxid = 0;
11197
11198     ipldbg(("ip_siocctl_get_muxid(%s:%u %p)\n",
11199         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11200     /*
11201      * Get the muxid saved in ill for I_PUNLINK.
11202      */
11203     arp_muxid = arl_get_muxid(ipif->ipif_ill);
11204     if (ipip->ipi_cmd_type == IF_CMD) {
11205         struct ifreq *ifr = (struct ifreq *)if_req;
11206
11207         ifr->ifr_ip_muxid = ipif->ipif_ill->ill_muxid;
11208         ifr->ifr_arp_muxid = arp_muxid;
11209     } else {
11210         struct lifreq *lifr = (struct lifreq *)if_req;
11211
11212         lifr->lifr_ip_muxid = ipif->ipif_ill->ill_muxid;
11213         lifr->lifr_arp_muxid = arp_muxid;
11214     }
11215     return (0);

```

```

11216 }

11218 /*
11219  * Set the subnet prefix. Does not modify the broadcast address.
11220  */
11221 /* ARGSUSED */
11222 int
11223 ip_ioctl_subnet(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11224 ip_ioctl_cmd_t *ipif, 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 (!V6_MASK_EQ(v6addr, v6mask, v6addr))
11267         return (EINVAL);

11269     if (IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6subnet, &v6addr) &&
11270 IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6net_mask, &v6mask))
11271         return (0); /* No change */

11273     if (ipif->ipif_flags & IPIF_UP) {
11274         /*
11275          * If the interface is already marked up,
11276          * we call ipif_down which will take care
11277          * of ditching any IRES that have been set
11278          * up based on the old interface address.
11279          */
11280         err = ipif_logical_down(ipif, q, mp);
11281         if (err == EINPROGRESS)

```

```

11282         return (err);
11283         (void) ipif_down_tail(ipif);
11284         need_up = B_TRUE;
11285     }

11287     err = ip_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 *ipif, void *if_req)
11330 {
11331     int addrlen;
11332     in6_addr_t v6addr;
11333     in6_addr_t v6mask;
11334     struct lifreq *lifr = (struct lifreq *)if_req;

11336     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 = lifr->lifr_addrlen;
11341     if (ipif->ipif_isv6) {
11342         sin6_t *sin6;

11344         sin6 = (sin6_t *)sin;
11345         v6addr = sin6->sin6_addr;
11346     } else {
11347         ipaddr_t addr;

```

```

11349         addr = sin->sin_addr.s_addr;
11350         IN6_IPADDR_TO_V4MAPPED(addr, &v6addr);
11351         addrlen += IPV6_ABITS - IP_ABITS;
11352     }
11353     (void) ip_plen_to_mask_v6(addrlen, &v6mask);
11354
11355     return (ip_ioctl_subnet_tail(ipif, v6addr, v6mask, q, mp, B_TRUE));
11356 }
11357
11358 /* ARGSUSED */
11359 int
11360 ip_ioctl_get_subnet(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11361     ip_ioctl_cmd_t *ipip, void *if_req)
11362 {
11363     struct lifreq *lifr = (struct lifreq *)if_req;
11364     struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)sin;
11365
11366     ipldbg(("ip_ioctl_get_subnet(%s:%u %p)\n",
11367         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11368     ASSERT(ipip->ipi_cmd_type == LIF_CMD);
11369
11370     if (ipif->ipif_isv6) {
11371         *sin6 = sin6_null;
11372         sin6->sin6_family = AF_INET6;
11373         sin6->sin6_addr = ipif->ipif_v6subnet;
11374         lifr->lifr_addrlen =
11375             ip_mask_to_plen_v6(&ipif->ipif_v6net_mask);
11376     } else {
11377         *sin = sin_null;
11378         sin->sin_family = AF_INET;
11379         sin->sin_addr.s_addr = ipif->ipif_subnet;
11380         lifr->lifr_addrlen = ip_mask_to_plen(ipif->ipif_net_mask);
11381     }
11382     return (0);
11383 }
11384
11385 /*
11386  * Set the IPv6 address token.
11387  */
11388 /* ARGSUSED */
11389 int
11390 ip_ioctl_token(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11391     ip_ioctl_cmd_t *ipi, void *if_req)
11392 {
11393     ill_t *ill = ipif->ipif_ill;
11394     int err;
11395     in6_addr_t v6addr;
11396     in6_addr_t v6mask;
11397     boolean_t need_up = B_FALSE;
11398     int i;
11399     sin6_t *sin6 = (sin6_t *)sin;
11400     struct lifreq *lifr = (struct lifreq *)if_req;
11401     int addrlen;
11402
11403     ipldbg(("ip_ioctl_token(%s:%u %p)\n",
11404         ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11405     ASSERT(IAM_WRITER_IPIF(ipif));
11406
11407     addrlen = lifr->lifr_addrlen;
11408     /* Only allow for logical unit zero i.e. not on "le0:17" */
11409     if (ipif->ipif_id != 0)
11410         return (EINVAL);
11411
11412     if (!ipif->ipif_isv6)
11413         return (EINVAL);

```

```

11415     if (addrlen > IPV6_ABITS)
11416         return (EINVAL);
11417
11418     v6addr = sin6->sin6_addr;
11419
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     }
11430
11431     if (V6_MASK_EQ(v6addr, v6mask, ill->ill_token) &&
11432         ill->ill_token_length == addrlen)
11433         return (0); /* No change */
11434
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_ioctl_token_tail(ipif, sin6, addrlen, q, mp, need_up);
11443     return (err);
11444 }
11445
11446 static int
11447 ip_ioctl_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;
11455
11456     ipldbg(("ip_ioctl_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;
11467
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;
11472
11473     /* Reconfigure the link-local address based on this new token */
11474     ipif_setlinklocal(ill->ill_ipif);
11475
11476     mutex_exit(&ill->ill_lock);
11477
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 *ipi, void *if_req)
11498 {
11499     ill_t *ill;
11500     sin6_t *sin6 = (sin6_t *)sin;
11501     struct lifreq *lifr = (struct lifreq *)if_req;

11503     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     lifr->lifr_addrlen = ill->ill_token_length;
11517     return (0);
11518 }

11520 /*
11521  * Set (hardware) link specific information that might override
11522  * what was acquired through the DL_INFO_ACK.
11523  */
11524 /* ARGSUSED */
11525 int
11526 ip_ioctl_lnkinfo(ipif_t *ipif, sin_t *sin, queue_t *q, mblk_t *mp,
11527 ip_ioctl_cmd_t *ipi, void *if_req)
11528 {
11529     ill_t *ill = ipif->ipif_ill;
11530     int ip_min_mtu;
11531     struct lifreq *lifr = (struct lifreq *)if_req;
11532     lif_ifinfo_req_t *lir;

11534     ipldbg(("ip_ioctl_lnkinfo(%s:%u %p)\n",
11535 ipif->ipif_ill->ill_name, ipif->ipif_id, (void *)ipif));
11536     lir = &lifr->lifr_ifinfo;
11537     ASSERT(IAM_WRITER_IPIF(ipif));

11539     /* Only allow for logical unit zero i.e. not on "bge0:17" */
11540     if (ipif->ipif_id != 0)
11541         return (EINVAL);

11543     /* Set interface MTU. */
11544     if (ipif->ipif_isv6)
11545         ip_min_mtu = IPV6_MIN_MTU;

```

```

11546     else
11547         ip_min_mtu = IP_MIN_MTU;

11549     /*
11550      * Verify values before we set anything. Allow zero to
11551      * mean unspecified.
11552      */
11553     * XXX We should be able to set the user-defined lir_mtu to some value
11554     * that is greater than ill_current_frag but less than ill_max_frag- the
11555     * ill_max_frag value tells us the max MTU that can be handled by the
11556     * datalink, whereas the ill_current_frag is dynamically computed for
11557     * some link-types like tunnels, based on the tunnel PMTU. However,
11558     * since there is currently no way of distinguishing between
11559     * administratively fixed link mtu values (e.g., those set via
11560     * /sbin/dladm) and dynamically discovered MTUs (e.g., those discovered
11561     * for tunnels) we conservatively choose the ill_current_frag as the
11562     * upper-bound.
11563     */
11564     if (lir->lir_maxmtu != 0 &&
11565         (lir->lir_maxmtu > ill->ill_current_frag ||
11566          lir->lir_maxmtu < ip_min_mtu))
11567         return (EINVAL);
11568     if (lir->lir_reachtime != 0 &&
11569         lir->lir_reachtime > ND_MAX_REACHTIME)
11570         return (EINVAL);
11571     if (lir->lir_reachretrans != 0 &&
11572         lir->lir_reachretrans > ND_MAX_REACHRETRANSTIME)
11573         return (EINVAL);

11575     mutex_enter(&ill->ill_lock);
11576     /*
11577      * The dce and fragmentation code can handle changes to ill_mtu
11578      * concurrent with sending/fragmenting packets.
11579      */
11580     if (lir->lir_maxmtu != 0)
11581         ill->ill_user_mtu = lir->lir_maxmtu;

11583     if (lir->lir_reachtime != 0)
11584         ill->ill_reachable_time = lir->lir_reachtime;

11586     if (lir->lir_reachretrans != 0)
11587         ill->ill_reachable_retrans_time = lir->lir_reachretrans;

11589     ill->ill_max_hops = lir->lir_maxhops;
11590     ill->ill_max_buf = ND_MAX_Q;
11591     if (!(ill->ill_flags & ILLF_FIXEDMTU) && ill->ill_user_mtu != 0) {
11592         /*
11593          * ill_mtu is the actual interface MTU, obtained as the min
11594          * of user-configured mtu and the value announced by the
11595          * driver (via DL_NOTE_SDU_SIZE/DL_INFO_ACK). Note that since
11596          * we have already made the choice of requiring
11597          * ill_user_mtu < ill_current_frag by the time we get here,
11598          * the ill_mtu effectively gets assigned to the ill_user_mtu
11599          * here.
11600          */
11601         ill->ill_mtu = MIN(ill->ill_current_frag, ill->ill_user_mtu);
11602         ill->ill_mc_mtu = MIN(ill->ill_mc_mtu, ill->ill_user_mtu);
11603     }
11604     mutex_exit(&ill->ill_lock);

11606     /*
11607      * Make sure all dce_generation checks find out
11608      * that ill_mtu/ill_mc_mtu has changed.
11609      */
11610     if (!(ill->ill_flags & ILLF_FIXEDMTU) && (lir->lir_maxmtu != 0))
11611         dce_increment_all_generations(ill->ill_isv6, ill->ill_ipst);

```

```

11613      /*
11614       * Refresh IPMP meta-interface MTU if necessary.
11615       */
11616      if (IS_UNDER_IPMP(ill))
11617          ipmp_illgrp_refresh_mtu(ill->ill_grp);
11619      return (0);
11620 }

11622 /* ARGSUSED */
11623 int
11624 ip_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_sioctl_copyin_setup
11760     */
11761     cr = msg_getcred(mp, NULL);
11762     if (cr == NULL)
11763         cr = iocp->ioc_cr;

11765     /*
11766     * Refhold the conn in case the request gets queued up in some lookup
11767     */
11768     ASSERT(CONN_Q(q));
11769     connp = Q_TO_CONN(q);
11770     CONN_INC_REF(connp);
11771     CONN_INC_IOCTLREF(connp);
11772     if (ipft->ipft_pfi &&
11773         ((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     greply(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 SIOCIFNAME 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 not link it to the ipmp_grp_t until
12062      * I_PLINK is called; see ip_ioctl_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 out:
12153     mutex_exit(&ill->ill_lock);
12154     return (ipif);
12155 }
12156
12157 /*
12158  * Remove the neighbor cache entries associated with this logical
12159  * interface.
12160  */
12161 int
12162 ipif_arp_down(ipif_t *ipif)
12163 {
12164     ill_t    *ill = ipif->ipif_ill;
12165     int      err = 0;
12166
12167     ipldbg(("ipif_arp_down(%s:%u)\n", ill->ill_name, ipif->ipif_id));
12168     ASSERT(IAM_WRITER_IPIF(ipif));
12169
12170     DTRACE_PROBE3(ipif_downup, char *, "ipif_arp_down",
12171         ill_t *, ill, ipif_t *, ipif);
12172     ipif_nce_down(ipif);
12173
12174     /*
12175      * If this is the last ipif that is going down and there are no
12176      * duplicate addresses we may yet attempt to re-probe, then we need to
12177      * clean up ARP completely.
12178      */
12179     if (ill->ill_ipif_up_count == 0 && ill->ill_ipif_dup_count == 0 &&
12180         !ill->ill_logical_down && ill->ill_net_type == IRE_IF_RESOLVER) {
12181         /*
12182          * If this was the last ipif on an IPMP interface, purge any
12183          * static ARP entries associated with it.
12184          */
12185         if (IS_IPMP(ill))
12186             ipmp_illgrp_refresh_arpent(ill->ill_grp);
12187
12188         /* UNBIND, DETACH */
12189         err = arp_ll_down(ill);
12190     }
12191
12192     return (err);
12193 }
12194
12195 /*
12196  * Get the resolver set up for a new IP address. (Always called as writer.)
12197  * Called both for IPv4 and IPv6 interfaces, though it only does some
12198  * basic DAD related initialization for IPv6. Honors ILLF_NOARP.
12199  */
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;
12217
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));
12221
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);
12247
12248     err = ipif_arp_up(ipif, res_act, was_dup);
12249     return (err);
12250 }
12251
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;
12262
12263     if (isv6) {
12264         ncec = ncec_lookup_illgrp_v6(ipif->ipif_ill,
12265             &ipif->ipif_v6lcl_addr);
12266     } else {
12267         ipaddr_t v4addr;
12268
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     ipldbg(("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_xop_mphead == NULL && ipx->ipx_mphead == NULL);
12385     ASSERT(ipx->ipx_pending_mp == NULL);
12386     kmem_free(ipsq, sizeof (ipsq_t));
12387 }

12389 static int
12390 ill_up_ipifs_on_ill(ill_t *ill, queue_t *q, mblk_t *mp)
12391 {
12392     int err = 0;
12393     ipif_t *ipif;

12395     if (ill == NULL)
12396         return (0);

12398     ASSERT(IAM_WRITER_ILL(ill));
12399     ill->ill_up_ipifs = B_TRUE;
12400     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
12401         if (ipif->ipif_was_up) {
12402             if (!(ipif->ipif_flags & IPIF_UP))
12403                 err = ipif_up(ipif, q, mp);

```

```

12404         ipif->ipif_was_up = B_FALSE;
12405         if (err != 0) {
12406             ASSERT(err == EINPROGRESS);
12407             return (err);
12408         }
12409     }
12410 }
12411 ill->ill_up_ipifs = B_FALSE;
12412 return (0);
12413 }

12415 /*
12416  * This function is called to bring up all the ipifs that were up before
12417  * bringing the ill down via ill_down_ipifs().
12418  */
12419 int
12420 ill_up_ipifs(ill_t *ill, queue_t *q, mblk_t *mp)
12421 {
12422     int err;

12424     ASSERT(IAM_WRITER_ILL(ill));

12426     if (ill->ill_replumbing) {
12427         ill->ill_replumbing = 0;
12428         /*
12429          * Send down REPLUMB_DONE notification followed by the
12430          * BIND_REQ on the arp stream.
12431          */
12432         if (!ill->ill_isv6)
12433             arp_send_replumb_conf(ill);
12434     }
12435     err = ill_up_ipifs_on_ill(ill->ill_phyint->phyint_illv4, q, mp);
12436     if (err != 0)
12437         return (err);

12439     return (ill_up_ipifs_on_ill(ill->ill_phyint->phyint_illv6, q, mp));
12440 }

12442 /*
12443  * Bring down any IPIF_UP ipifs on ill. If "logical" is B_TRUE, we bring
12444  * down the ipifs without sending DL_UNBIND_REQ to the driver.
12445  */
12446 static void
12447 ill_down_ipifs(ill_t *ill, boolean_t logical)
12448 {
12449     ipif_t *ipif;

12451     ASSERT(IAM_WRITER_ILL(ill));

12453     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
12454         /*
12455          * We go through the ipif_down logic even if the ipif
12456          * is already down, since routes can be added based
12457          * on down ipifs. Going through ipif_down once again
12458          * will delete any IRES created based on these routes.
12459          */
12460         if (ipif->ipif_flags & IPIF_UP)
12461             ipif->ipif_was_up = B_TRUE;

12463         if (logical) {
12464             (void) ipif_logical_down(ipif, NULL, NULL);
12465             ipif_non_duplicate(ipif);
12466             (void) ipif_down_tail(ipif);
12467         } else {
12468             (void) ipif_down(ipif, NULL, NULL);
12469         }

```

```

12470     }
12471 }

12473 /*
12474  * Redo source address selection. This makes IXAF_VERIFY_SOURCE take
12475  * a look again at valid source addresses.
12476  * This should be called each time after the set of source addresses has been
12477  * changed.
12478  */
12479 void
12480 ip_update_source_selection(ip_stack_t *ipst)
12481 {
12482     /* We skip past SRC_GENERATION_VERIFY */
12483     if (atomic_add_32_nv(&ipst->ips_src_generation, 1) ==
12484         SRC_GENERATION_VERIFY)
12485         atomic_add_32(&ipst->ips_src_generation, 1);
12486 }

12488 /*
12489  * Finish the group join started in ip_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 SIOCSLIFGROUPNAME 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 *liffr = ifreq;
12528     ill_t      *ill = ipif->ipif_ill;
12529     ip_stack_t   *ipst = ill->ill_ipst;
12530     phyint_t     *phyi = ill->ill_phyint;
12531     ipmp_grp_t    *grp = phyi->phyint_grp;
12532     mblk_t        *ipsq_mp;
12533     int           err = 0;

12535     /*

```

```

12536     * Note that phyint_grp can only change here, where we're exclusive.
12537     */
12538     ASSERT(IAM_WRITER_ILL(ill));

12540     if (ipif->ipif_id != 0 || ill->ill_usesrc_grp_next != NULL ||
12541         (phyi->phyint_flags & PHYI_VIRTUAL))
12542         return (EINVAL);

12544     lifr->lifr_groupname[LIFGRNAMSIZ - 1] = '\0';

12546     rw_enter(&ipst->ips_ipmp_lock, RW_WRITER);

12548     /*
12549      * If the name hasn't changed, there's nothing to do.
12550      */
12551     if (grp != NULL && strcmp(grp->gr_name, lifr->lifr_groupname) == 0)
12552         goto unlock;

12554     /*
12555      * Handle requests to rename an IPMP meta-interface.
12556      *
12557      * Note that creation of the IPMP meta-interface is handled in
12558      * userland through the standard plumbing sequence. As part of the
12559      * plumbing the IPMP meta-interface, its initial groupname is set to
12560      * the name of the interface (see ipif_set_values_tail()).
12561      */
12562     if (IS_IPMP(ill)) {
12563         err = ipmp_grp_rename(grp, lifr->lifr_groupname);
12564         goto unlock;
12565     }

12567     /*
12568      * Handle requests to add or remove an IP interface from a group.
12569      */
12570     if (lifr->lifr_groupname[0] != '\0') { /* add */
12571         /*
12572          * Moves are handled by first removing the interface from
12573          * its existing group, and then adding it to another group.
12574          * So, fail if it's already in a group.
12575          */
12576         if (IS_UNDER_IPMP(ill)) {
12577             err = EALREADY;
12578             goto unlock;
12579         }

12581         grp = ipmp_grp_lookup(lifr->lifr_groupname, ipst);
12582         if (grp == NULL) {
12583             err = ENOENT;
12584             goto unlock;
12585         }

12587         /*
12588          * Check if the phyint and its ill 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 ill 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_refhold(ill);
12621     qwriter_ip(ill, ill->ill_rq, ipsq_mp, ip_join_illgrps,
12622         SWITCH_OP, B_FALSE);
12623     return (0);
12624 } else {
12625     /*
12626      * Request to remove the interface from a group. If the
12627      * interface is not in a group, this trivially succeeds.
12628      */
12629     rw_exit(&ipst->ips_ipmp_lock);
12630     if (IS_UNDER_IPMP(ill))
12631         ipmp_phyint_leave_grp(phyi);
12632     return (0);
12633 }
12634 unlock:
12635     rw_exit(&ipst->ips_ipmp_lock);
12636     return (err);
12637 }

12639 /*
12640  * Process an SIOCGLIFBINDING request.
12641  */
12642 /* ARGSUSED */
12643 int
12644 ip_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    *lifrr = 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         lifrr->lifr_groupname[0] = '\0';
12678     else
12679         (void) strcpy(lifrr->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_refhold(ill);
12848         /* qwriter_ip() does the ill_refrele() */
12849         qwriter_ip(ill, ill->ill_wq, mp, ill_dlpi_send_writer,
12850                 NEW_OP, B_TRUE);
12851         return;
12852     }

12854     mutex_enter(&ill->ill_lock);
12855     if (ill->ill_dlpi_pending != DL_PRIM_INVAL) {
12856         /* Must queue message. Tail insertion */
12857         mpp = &ill->ill_dlpi_deferred;
12858         while (*mpp != NULL)
12859             mpp = &((*mpp)->b_next);

12861         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 = nexttmp) {
12934         nexttmp = 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/mlld_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 ixa_ifindex. If IPMP it will be the lower ill which
13193      * makes ip_select_route pick the IRE_MULTICAST for the cast_ill.
13194      * That is necessary to handle IGMP/MLD snooping switches.
13195      */

```

```

13196     ixas.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->b_rptr;

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->b_rptr;

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_ncac_cnt; Number of ncacs 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_ncac_cnt is the number of ncac's referencing the ill thru ncac_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 ncac_inactive() where the ncac
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 ioctl's 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 ioctl's 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) untimout(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) untimout(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_strtol(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          if (! (IPIF_IS_CHANGING(ipif) ||
13919                IPIF_IS_CONDEMNED(ipif) ||
13920                IAM_WRITER_IPIF(ipif))) {

```

```

13922      ipif_refhold_locked(ipif);
13923      mutex_exit(&ill->ill_lock);
13924      /*
13925       * Drop locks before calling ill_refrele
13926       * since it can potentially call into
13927       * ipif_ill_refrele_tail which can end up
13928       * in trying to acquire any lock.
13929       */
13930      RELEASE_CONN_LOCK(q);
13931      ill_refrele(ill);
13932      return (ipif);
13933  } else if (q != NULL && !IPIF_IS_CONDEMNED(ipif)) {
13934      ipsq = ill->ill_phyint->phyint_ipsq;
13935      mutex_enter(&ipsq->ipsq_lock);
13936      mutex_enter(&ipsq->ipsq_xop->ipx_lock);
13937      mutex_exit(&ill->ill_lock);
13938      ipsq_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_t *icmp;
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_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     icmp_h = (icmp_h_t *)&ipha[1];
14002     icmp_h->icmp_h_type = ICMP_ADDRESS_MASK_REPLY;
14003     bcopy(&ipif->ipif_net_mask, &icmp_h[1], IP_ADDR_LEN);
14004     icmp_h->icmp_h_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     ixas_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;
14253
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;
14270
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 }
14290
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;
14306
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));
14310
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);
14315
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);
14322
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;
14332
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     }
14350
14351     return (0);
14352 }
14353
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;
14369
14370     ASSERT(IAM_WRITER_IPIF(ipif));
14371
14372     ipldbg(("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);
14375
14376     /* Shouldn't get here if it is already up. */
14377     if (ipif->ipif_flags & IPIF_UP)
14378         return (EALREADY);
14379
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 defences 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);

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);

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          }

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          }

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;

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);

14467      /*
14468       * Tell SCTP that the ipif has moved. Note that even if we
14469       * had to allocate a new ipif, the original sequence id was
14470       * preserved and therefore SCTP won't know.
14471       */
14472      sctp_move_ipif(ipif, ill, ipmp_ill);

14474      /*
14475       * If the ipif being brought up was on slot zero, then we
14476       * first need to bring up the placeholder we stuck there. In
14477       * ip_rput_dlpi_writer(), arp_bringup_done(), or the recursive
14478       * call to ipif_up() itself, if we successfully bring up the
14479       * placeholder, we'll check ill_move_ipif and bring it up too.
14480       */
14481      if (ipif_orig_id == 0) {
14482          ASSERT(ill->ill_move_ipif == NULL);
14483          ill->ill_move_ipif = ipif;
14484          if ((err = ipif_up(ill->ill_ipif, q, mp)) == 0)
14485              ASSERT(ill->ill_move_ipif == NULL);
14486          if (err != EINPROGRESS)
14487              ill->ill_move_ipif = NULL;
14488          return (err);
14489      }

14491      /*
14492       * Bring it up on the IPMP ill.
14493       */
14494      return (ipif_up(ipif, q, mp));
14495  }

14497      /* Skip arp/ndp for any loopback interface. */
14498      if (ill->ill_wq != NULL) {
14499          conn_t *connp = CONN_Q(q) ? Q_TO_CONN(q) : NULL;
14500          ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;

14502          if (!ill->ill_dl_up) {
14503              /*
14504               * ill_dl_up is not yet set. i.e. we are yet to
14505               * DL_BIND with the driver and this is the first
14506               * logical interface on the ill to become "up".
14507               * Tell the driver to get going (via DL_BIND REQ).
14508               * Note that changing "significant" IFF flags
14509               * address/netmask etc cause a down/up dance, but
14510               * does not cause an unbind (DL_UNBIND) with the driver
14511               */
14512              return (ill_dl_up(ill, ipif, mp, q));
14513          }

14515      /*

```

```

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     */
14524
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);
14535
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     }
14545
14546     if (isv6 && err == 0)
14547         err = ipif_ndp_up(ipif, B_TRUE);
14548
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 }
14566
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 }
14575
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);
14586
14587     if (ill->ill_ire_multicast != NULL)
14588         return (0);
14589
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);
14602
14603     ill->ill_ire_multicast = ire;
14604     return (0);
14605 }
14606
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 }
14622
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;
14637
14638     DTRACE_PROBE2(ill_downup, char *, "ill_dl_up", ill_t *, ill);
14639
14640     ipldbg(("ill_dl_up(%s)\n", ill->ill_name));
14641     ASSERT(IAM_WRITER_ILL(ill));
14642     ASSERT(mp != NULL);
14643
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 (dhcpcinit() != 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);
14781
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);
14785
14786         /*
14787          * lo0:1 and subsequent ipifs were marked IRE_LOCAL in
14788          * ipif_lookup_on_name(), but in the case of zones we can have
14789          * several loopback addresses on lo0. So all the interfaces with
14790          * loopback addresses need to be marked IRE_LOOPBACK.
14791          */
14792         if (V4_PART_OF_V6(ipif->ipif_v6lcl_addr) ==
14793             htonl(INADDR_LOOPBACK))
14794             ipif->ipif_ire_type = IRE_LOOPBACK;
14795         else
14796             ipif->ipif_ire_type = IRE_LOCAL;
14797         if (ill->ill_net_type != IRE_LOOPBACK)
14798             flags |= NCE_F_PUBLISH;
14799
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     }
14815
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     }
14840
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     }
14861
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);
14868
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     }
14876
14877     /* Join the allhosts multicast address */
14878     ipif_multicast_up(ipif);
14879
14880     if (!loopback && !update_src_selection &&
14881         !(ipif->ipif_flags & (IPIF_NOLOCAL|IPIF_ANYCAST|IPIF_DEPRECATED)))
14882         ip_update_source_selection(ill->ill_ipst);
14883
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);
14890
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 }
14901
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       * ill's 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;

15145              ire = ire_ftable_lookup_v4(ipif->ipif_brd_addr, 0, 0,
15146              IRE_BROADCAST, ipif->ipif_ill, ALL_ZONES, NULL,
15147              (MATCH_IRE_TYPE | MATCH_IRE_ILL), 0, ipst, NULL);

15149              if (ire == NULL) {
15150                  /*
15151                   * If there isn't a matching broadcast IRE,
15152                   * revert to the default for this netmask.
15153                   */
15154                  ipif->ipif_v6brd_addr = ipv6_all_zeros;
15155                  mutex_enter(&ipif->ipif_ill->ill_lock);
15156                  ipif_set_default(ipif);
15157                  mutex_exit(&ipif->ipif_ill->ill_lock);
15158              } else {
15159                  ire_refrele(ire);
15160              }
15161          }
15162      }
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 loose the count due to
15218          * a down/up dance.
15219          */
15220         atomic_add_32(&ipif->ipif_ib_pkt_count, ire->ire_ib_pkt_count);

15222         ire_delete(ire);
15223         ire_refrele_notr(ire);
15224     }
15225     rw_enter(&ipst->ips_ill_g_lock, RW_WRITER);
15226     ire = ipif->ipif_ire_if;
15227     ipif->ipif_ire_if = NULL;
15228     rw_exit(&ipst->ips_ill_g_lock);
15229     if (ire != NULL) {
15230         ire_delete(ire);
15231         ire_refrele_notr(ire);
15232     }
15234     /*
15235     * Delete the broadcast IRES.
15236     */
15237     if ((ipif->ipif_flags & IPIF_BROADCAST) &&
15238         !(ipif->ipif_flags & IPIF_NOXMIT))
15239         ipif_delete_bcast_ires(ipif);
15240 }

```

```

15242 /*
15243  * Checks for 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_rhdp, *dst_rhdp;
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_rhdp = NULL;
15378     if (is_system_labeled()) {
15379         dst_rhdp = find_tpc(&dst, IPV4_VERSION, B_FALSE);
15380         if (dst_rhdp == NULL)
15381             return (NULL);
15382         if (dst_rhdp->tpc_tp.host_type != UNLABELED) {
15383             TPC_RELE(dst_rhdp);
15384             dst_rhdp = 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_rhdp != NULL) {
15452          boolean_t incompat;

15454          src_rhdp = find_tpc(&ipif->ipif_lcl_addr,
15455                             IPV4_VERSION, B_FALSE);
15456          if (src_rhdp == NULL)
15457              continue;
15458          incompat = src_rhdp->tpc_tp.host_type != SUN_CIPSO ||
15459                   src_rhdp->tpc_tp.tp_doi !=
15460                   dst_rhdp->tpc_tp.tp_doi ||
15461                   (!blinrange(&dst_rhdp->tpc_tp.tp_def_label,
15462                               &src_rhdp->tpc_tp.tp_sl_range_cipso) &&
15463                    !blinlset(&dst_rhdp->tpc_tp.tp_def_label,
15464                              src_rhdp->tpc_tp.tp_sl_set_cipso));
15465          TPC_RELE(src_rhdp);
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_rhdp != NULL)
15522          TPC_RELE(dst_rhdp);

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 *srcp,
15579 uint32_t *generation, uint64_t *flagsp)
15580 {
15581     ipif_t *ipif;
15582     boolean_t notready = B_FALSE; /* Set if !ipif_addr_ready found */
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         *srcp = 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         *srcp = setsrc;
15606         return (0);
15607     }
15608     ipif = ipif_select_source_v4(ill, dst, zoneid, B_TRUE, &notready);
15609     if (ipif == NULL) {
15610         if (notready)
15611             return (ENETDOWN);
15612         else
15613             return (EADDRNOTAVAIL);
15614     }
15615     *srcp = ipif->ipif_lcl_addr;
15616     if (flagsp != NULL)
15617         *flagsp = ipif->ipif_flags;
15618     ipif_refrele(ipif);
15619     return (0);
15620 }
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 *ipip, void *dummy_ifreq)
15626 {
15627     /*
15628      * ill_phyint_reinit merged the v4 and v6 into a single
15629      * ipsq. We might not have been able to complete the
15630      * operation in ipif_set_values, if we could not become
15631      * exclusive. If so restart it here.
15632      */
15633     return (ipif_set_values_tail(ipif->ipif_ill, ipif, mp, q));
15634 }
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 *ipip, void *dummy_ifreq)
15644 {
15645     queue_t *q1 = q;
15646     char *cp;
15647     char interf_name[LIFNAMSIZ];
15648     uint_t ppa = *(uint_t *)mp->b_cont->b_cont->b_rptr;
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_sioctl_sifname(ipif_t *dummy_ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
15675 ip_ioctl_cmd_t *ipip, 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 queueing it on
15809  * ipx_pending_mp and the ioctl will complete in ip_rput.
15810  *
15811  * Can operate on either a module or a driver queue.
15812  * Returns an error if not a module queue.
15813  */
15814 /* ARGSUSED */
15815 int
15816 ip_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 *lifr = if_req;
15823     uint64_t   new_flags;

15825     ASSERT(ipif != NULL);
15826     ipldbg(("ip_ioctl_slifname %s\n", lifr->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 ill's, 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              ipldbg(("if_siocltl_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          ipldbg(("ip_siocltl_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          ipldbg(("ip_siocltl_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_siocltl_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_siocctl_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_siocctl_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_siocctl_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_siocctl_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_siocctl_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_siocctl_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 SIOCSLIFZONE to put hme0:1 in zone "myzone";
16207       * 2) hme0:1 is up and can't be brought down right away;
16208       * ip_sioctl_slifzone() returns EINPROGRESS and the request is queued;
16209       * 3) zone "myzone" is halted; the zone status switches to
16210       * 'shutting_down' and the zones framework sends SIOCGLIFCONF to list
16211       * the interfaces to remove - hme0:1 is not returned because it's not
16212       * yet in "myzone", so it won't be removed;
16213       * 4) the restart function for SIOCSLIFZONE is called; without the
16214       * status check here, we would have hme0:1 in "myzone" after it's been
16215       * destroyed.
16216       * Note that if the status check fails, we need to bring the interface
16217       * back to its state prior to ip_sioctl_slifzone(), hence the call to
16218       * ipif_up_done[v6]() .
16219       */
16220      status = ZONE_IS_UNINITIALIZED;
16221      if ((zptr = zone_find_by_id(zoneid)) != NULL) {
16222          status = zone_status_get(zptr);
16223          zone_rele(zptr);
16224      }
16225      if (status != ZONE_IS_READY && status != ZONE_IS_RUNNING) {
16226          if (ipif->ipif_isv6) {
16227              (void) ipif_up_done_v6(ipif);
16228          } else {
16229              (void) ipif_up_done(ipif);
16230          }
16231          return (EINVAL);

```

```

16232     }
16233
16234     (void) ipif_down_tail(ipif);
16235
16236     return (ip_ioctl_slifzone_tail(ipif, lifr->lifr_zoneid, q, mp,
16237         B_TRUE));
16238 }
16239
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;
16249
16250     ASSERT(IAM_WRITER_ILL(ill));
16251
16252     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next)
16253         if ((ipif->ipif_flags & set) && !(ipif->ipif_flags & clear))
16254             cnt++;
16255
16256     return (cnt);
16257 }
16258
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 }
16269
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 }
16278
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;
16285
16286     ASSERT(q->q_next == NULL);
16287     ASSERT(CONN_Q(q));
16288
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));
16293
16294     return (0);
16295 }
16296
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;
16302
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 }
16309
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;
16330
16331     ASSERT(RW_WRITE_HELD(&ipst->ips_ill_g_usesrc_lock));
16332     next_ill = uill->ill_usesrc_grp_next;
16333
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 }
16344
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;
16353
16354     ASSERT((ucill != NULL) && (ucill->ill_usesrc_grp_next != NULL) &&
16355         (uill != NULL) && RW_WRITE_HELD(&ipst->ips_ill_g_usesrc_lock));
16356
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_usersrc_ifindex != 0)) {
16365             return (-1);
16366         }
16368         ill = ill_prev_usersrc(ucill);
16369         ASSERT(ill->ill_usersrc_grp_next != NULL);
16371         /* Remove from the current list */
16372         if (ill->ill_usersrc_grp_next->ill_usersrc_grp_next == ill) {
16373             /* Only two elements in the list */
16374             ASSERT(ill->ill_usersrc_ifindex == 0);
16375             ill->ill_usersrc_grp_next = NULL;
16376         } else {
16377             ill->ill_usersrc_grp_next = ucill->ill_usersrc_grp_next;
16378         }
16380         if (ifindex == 0) {
16381             ucill->ill_usersrc_ifindex = 0;
16382             ucill->ill_usersrc_grp_next = NULL;
16383             return (0);
16384         }
16386         ucill->ill_usersrc_ifindex = ifindex;
16387         tmp_ill = uill->ill_usersrc_grp_next;
16388         uill->ill_usersrc_grp_next = ucill;
16389         ucill->ill_usersrc_grp_next =
16390             (tmp_ill != NULL) ? tmp_ill : uill;
16391         return (0);
16392     }
16394 /*
16395  * Set the ill_usersrc and ill_usersrc_head fields. See synchronization notes in
16396  * ip.c for locking details.
16397  */
16398 /* ARGSUSED */
16399 int
16400 ip_siocctl_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 *usersrc_ill, *usersrc_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 (usersrc_cli_ill->ill_usersrc_grp_next == NULL) {
16420             /* non usersrc group interface, nothing to reset */
16421             return (0);
16422         }
16423         ifindex = usersrc_cli_ill->ill_usersrc_ifindex;
16424         /* valid reset request */
16425         reset_flg = B_TRUE;
16426     }
16428     usersrc_ill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
16429     if (usersrc_ill == NULL)

```

```

16430         return (ENXIO);
16431         if (usersrc_ill == ipif->ipif_ill) {
16432             ill_refrele(usersrc_ill);
16433             return (EINVAL);
16434         }
16436         ipsq = ipsq_try_enter(NULL, usersrc_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 usersrc ILL */
16441             goto done;
16442         }
16444         /* USESRC isn't currently supported with IPMP */
16445         if (IS_IPMP(usersrc_ill) || IS_UNDER_IPMP(usersrc_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 (usersrc_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 usersrc_ill or a usersrc_ill is
16462          * already a client then return EINVAL
16463          */
16464         if (IS_USESRC_ILL(usersrc_cli_ill) || IS_USESRC_CLI_ILL(usersrc_ill)) {
16465             err = EINVAL;
16466             goto done;
16467         }
16469         /*
16470          * If the ill_usersrc_ifindex field is already set to what it needs to
16471          * be then this is a duplicate operation.
16472          */
16473         if (!reset_flg && usersrc_cli_ill->ill_usersrc_ifindex == ifindex) {
16474             err = 0;
16475             goto done;
16476         }
16478         ipldbg(("ip_siocctl_slifusesrc: usersrc_cli_ill %s, usersrc_ill %s,"
16479             " v6 = %d", usersrc_cli_ill->ill_name, usersrc_ill->ill_name,
16480             usersrc_ill->ill_isv6));
16482         /*
16483          * ill_g_usersrc_lock global lock protects the ill_usersrc_grp_next
16484          * and the ill_usersrc_ifindex fields
16485          */
16486         rw_enter(&ipst->ips_ill_g_usersrc_lock, RW_WRITER);
16488         if (reset_flg) {
16489             ret = ill_relink_usersrc_ills(usersrc_cli_ill, usersrc_ill, 0);
16490             if (ret != 0) {
16491                 err = EINVAL;
16492             }
16493             rw_exit(&ipst->ips_ill_g_usersrc_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 *ipip, void *if_req)
16541 {
16542     struct lifreq *lifr = (struct lifreq *)if_req;
16543     ill_t *ill = ipif->ipif_ill;

16545     /*
16546      * Need a lock since IFF_UP can be set even when there are
16547      * references to the ipif.
16548      */
16549     mutex_enter(&ill->ill_lock);
16550     if ((ipif->ipif_flags & IPIF_UP) && ipif->ipif_addr_ready == 0)
16551         lifr->lifr_dadstate = DAD_IN_PROGRESS;
16552     else
16553         lifr->lifr_dadstate = DAD_DONE;
16554     mutex_exit(&ill->ill_lock);
16555     return (0);
16556 }

16558 /*
16559  * comparison function used by avl.
16560  */
16561 static int

```

```

16562 ill_phyint_compare_index(const void *index_ptr, const void *phyip)
16563 {
16565     uint_t index;

16567     ASSERT(phyip != NULL && index_ptr != NULL);

16569     index = *((uint_t *)index_ptr);
16570     /*
16571      * let the phyint with the lowest index be on top.
16572      */
16573     if (((phyint_t *)phyip)->phyint_ifindex < index)
16574         return (1);
16575     if (((phyint_t *)phyip)->phyint_ifindex > index)
16576         return (-1);
16577     return (0);
16578 }

16580 /*
16581  * comparison function used by avl.
16582  */
16583 static int
16584 ill_phyint_compare_name(const void *name_ptr, const void *phyip)
16585 {
16586     ill_t *ill;
16587     int res = 0;

16589     ASSERT(phyip != NULL && name_ptr != NULL);

16591     if (((phyint_t *)phyip)->phyint_illv4)
16592         ill = ((phyint_t *)phyip)->phyint_illv4;
16593     else
16594         ill = ((phyint_t *)phyip)->phyint_illv6;
16595     ASSERT(ill != NULL);

16597     res = strcmp(ill->ill_name, (char *)name_ptr);
16598     if (res > 0)
16599         return (1);
16600     else if (res < 0)
16601         return (-1);
16602     return (0);
16603 }

16605 /*
16606  * This function is called on the unplumb path via ill_glist_delete() when
16607  * there are no ill's 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) strcpy(ill->ill_phyint->phyint_name, ill->ill_name, LIFNAMSIZ);

16678     phyi = avl_find(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
16679         ill->ill_name, &where);

16681     /*
16682     * 1. We grabbed the ill_g_lock before inserting this ill into
16683     * the global list of ills. So no other thread could have located
16684     * this ill and hence the ipsq of this ill is guaranteed to be empty.
16685     * 2. Now locate the other protocol instance of this ill.
16686     * 3. Now grab both ill locks in the right order, and the phyint lock of
16687     * the new ipsq. Holding ill locks + ill_g_lock ensures that the ipsq
16688     * of neither ill can change.
16689     * 4. Merge the phyint and thus the ipsq as well of this ill onto the
16690     * other ill.
16691     * 5. Release all locks.
16692     */

```

```

16694     /*
16695     * Look for IPv4 if we are initializing IPv6 or look for IPv6 if
16696     * we are initializing IPv4.
16697     */
16698     if (phyi != NULL) {
16699         ill_other = (isv6) ? phyi->phyint_illv4 : phyi->phyint_illv6;
16700         ASSERT(ill_other->ill_phyint != NULL);
16701         ASSERT((isv6 && !ill_other->ill_isv6) ||
16702             (!isv6 && ill_other->ill_isv6));
16703         GRAB_ILL_LOCKS(ill, ill_other);
16704         /*
16705         * We are potentially throwing away phyint_flags which
16706         * could be different from the one that we obtain from
16707         * ill_other->ill_phyint. But it is okay as we are assuming
16708         * that the state maintained within IP is correct.
16709         */
16710         mutex_enter(&phyi->phyint_lock);
16711         if (isv6) {
16712             ASSERT(phyi->phyint_illv6 == NULL);
16713             phyi->phyint_illv6 = ill;
16714         } else {
16715             ASSERT(phyi->phyint_illv4 == NULL);
16716             phyi->phyint_illv4 = ill;
16717         }

16719         /*
16720         * Delete the old phyint and make its ipsq eligible
16721         * to be freed in ipsq_exit().
16722         */
16723         phyi_old->phyint_illv4 = NULL;
16724         phyi_old->phyint_illv6 = NULL;
16725         phyi_old->phyint_ipsq->ipsq_phyint = NULL;
16726         phyi_old->phyint_name[0] = '\0';
16727         mi_free(phyi_old);
16728     } else {
16729         mutex_enter(&ill->ill_lock);
16730         /*
16731         * We don't need to acquire any lock, since
16732         * the ill is not yet visible globally and we
16733         * have not yet released the ill_g_lock.
16734         */
16735         phyi = phyi_old;
16736         mutex_enter(&phyi->phyint_lock);
16737         /* XXX We need a recovery strategy here. */
16738         if (!phyint_assign_ifindex(phyi, ipst))
16739             cmn_err(CE_PANIC, "phyint_assign_ifindex() failed");

16741         avl_insert(&ipst->ips_phyint_g_list->phyint_list_avl_by_name,
16742             (void *)phyi, where);

16744         (void) avl_find(&ipst->ips_phyint_g_list->
16745             phyint_list_avl_by_index,
16746             &phyi->phyint_ifindex, &where);
16747         avl_insert(&ipst->ips_phyint_g_list->phyint_list_avl_by_index,
16748             (void *)phyi, where);
16749     }

16751     /*
16752     * Reassigning ill_phyint automatically reassigns the ipsq also.
16753     * pending mp is not affected because that is per ill basis.
16754     */
16755     ill->ill_phyint = phyi;

16757     /*
16758     * Now that the phyint's ifindex has been assigned, complete the
16759     * remaining

```

```

16760      */
16761      ill->ill_ip_mib->ipIfStatsIfIndex = ill->ill_phyint->phyint_ifindex;
16762      if (ill->ill_isv6) {
16763          ill->ill_icmp6_mib->ipV6IfIcmpIfIndex =
16764              ill->ill_phyint->phyint_ifindex;
16765          ill->ill_mcast_type = ipst->ips_mld_max_version;
16766      } else {
16767          ill->ill_mcast_type = ipst->ips_igmp_max_version;
16768      }
16770      /*
16771       * Generate an event within the hooks framework to indicate that
16772       * a new interface has just been added to IP. For this event to
16773       * be generated, the network interface must, at least, have an
16774       * ifindex assigned to it. (We don't generate the event for
16775       * loopback since ill_lookup_on_name() has its own NE_PLUMB event.)
16776       *
16777       * This needs to be run inside the ill_g_lock perimeter to ensure
16778       * that the ordering of delivered events to listeners matches the
16779       * order of them in the kernel.
16780       */
16781      if (!IS_LOOPBACK(ill)) {
16782          ill_nic_event_dispatch(ill, 0, NE_PLUMB, ill->ill_name,
16783              ill->ill_name_length);
16784      }
16785      RELEASE_ILL_LOCKS(ill, ill_other);
16786      mutex_exit(&phyi->phyint_lock);
16787  }
16789  /*
16790   * Notify any downstream modules of the name of this interface.
16791   * An M_IOCTL is used even though we don't expect a successful reply.
16792   * Any reply message from the driver (presumably an M_IOCNAK) will
16793   * eventually get discarded somewhere upstream. The message format is
16794   * simply an SIOCSLIFNAME ioctl just as might be sent from ifconfig
16795   * to IP.
16796   */
16797  static void
16798  ip_ifname_notify(ill_t *ill, queue_t *q)
16799  {
16800      mblk_t *mpl, *mp2;
16801      struct iocblk *iocp;
16802      struct lifreq *lifr;
16804      mpl = mkiocb(SIOCSLIFNAME);
16805      if (mpl == NULL)
16806          return;
16807      mp2 = allocb(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 interf_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 chosen 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 ETHERADDR 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 == ETHERADDR) {
17398         /* Form EUI-64 like address */
17399         addr = (char *)&v6addr->s6_addr32[2];
17400         bcopy(ill->ill_phys_addr, addr, 3);
17401         addr[0] ^= 0x2; /* Toggle Universal/Local bit */
17402         addr[3] = (char)0xff;
17403         addr[4] = (char)0xfe;
17404         bcopy(ill->ill_phys_addr + 3, addr + 5, 3);
17405     }
17406 }

17408 /* ARGSUSED */
17409 static void
17410 ip_nodef_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17411 {
17412 }

17414 typedef struct ipmp_ifcookie {
17415     uint32_t ic_hostid;
17416     char ic_ifname[LIFNAMSIZ];
17417     char ic_zonename[ZONENAME_MAX];
17418 } ipmp_ifcookie_t;

```

```

17420 /*
17421  * Construct a pseudo-random interface ID for the IPMP interface that's both
17422  * predictable and (almost) guaranteed to be unique.
17423  */
17424 static void
17425 ip_ipmp_v6intfid(ill_t *ill, in6_addr_t *v6addr)
17426 {
17427     zone_t      *zp;
17428     uint8_t      *addr;
17429     uchar_t      hash[16];
17430     ulong_t      hostid;
17431     MD5_CTX      ctx;
17432     ipmp_ifcookie_t ic = { 0 };
17433
17434     ASSERT(IS_IPMP(ill));
17435
17436     (void) ddi_strtoul(hw_serial, NULL, 10, &hostid);
17437     ic.ic_hostid = htonl((uint32_t)hostid);
17438
17439     (void) strcpy(ic.ic_ifname, ill->ill_name, LIFNAMSIZ);
17440
17441     if ((zp = zone_find_by_id(ill->ill_zoneid)) != NULL) {
17442         (void) strcpy(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_rptr;
17522     if (ill->ill_sap_length < 0) {
17523         bphys_addr = (uchar_t *)dlur +
17524             dlur->dl_dest_addr_offset;
17525     } else {
17526         bphys_addr = (uchar_t *)dlur +
17527             dlur->dl_dest_addr_offset + ill->ill_sap_length;
17528     }
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     ill = ill_lookup_on_ifindex(index, isv6, ipst);
17703     if (ill == NULL)
17704         return (NULL);

17706     mutex_enter(&ill->ill_lock);
17707     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17708         if (IPIF_IS_CONDEMNED(ipif))
17709             continue;
17710         if (zoneid != ALL_ZONES && ipif->ipif_zoneid != zoneid &&
17711             ipif->ipif_zoneid != ALL_ZONES)
17712             continue;

17714         mutex_exit(&ill->ill_lock);
17715         return (ill);
17716     }
17717     mutex_exit(&ill->ill_lock);
17718     ill_refrele(ill);
17719     return (NULL);
17720 }

17722 /*
17723  * Return a pointer to an ipif_t given a combination of (ill_idx, ipif_id)
17724  * If a pointer to an ipif_t is returned then the caller will need to do
17725  * an ill_refrele().
17726  */
17727 ipif_t *
17728 ipif_getby_indexes(uint_t ifindex, uint_t lifidx, boolean_t isv6,
17729     ip_stack_t *ipst)
17730 {
17731     ipif_t *ipif;
17732     ill_t *ill;

17734     ill = ill_lookup_on_ifindex(ifindex, isv6, ipst);
17735     if (ill == NULL)
17736         return (NULL);

17738     mutex_enter(&ill->ill_lock);
17739     if (ill->ill_state_flags & ILL_CONDEMNED) {
17740         mutex_exit(&ill->ill_lock);
17741         ill_refrele(ill);
17742         return (NULL);
17743     }

17745     for (ipif = ill->ill_ipif; ipif != NULL; ipif = ipif->ipif_next) {
17746         if (!IPIF_CAN_LOOKUP(ipif))
17747             continue;
17748         if (lifidx == ipif->ipif_id) {
17749             ipif_refhold_locked(ipif);

```

```

17750         break;
17751     }
17752 }

17754     mutex_exit(&ill->ill_lock);
17755     ill_refrele(ill);
17756     return (ipif);
17757 }

17759 /*
17760  * Set ill_inputfn based on the current know state.
17761  * This needs to be called when any of the factors taken into
17762  * account changes.
17763  */
17764 void
17765 ill_set_inputfn(ill_t *ill)
17766 {
17767     ip_stack_t *ipst = ill->ill_ipst;

17769     if (ill->ill_isv6) {
17770         if (is_system_labeled())
17771             ill->ill_inputfn = ill_input_full_v6;
17772         else
17773             ill->ill_inputfn = ill_input_short_v6;
17774     } else {
17775         if (is_system_labeled())
17776             ill->ill_inputfn = ill_input_full_v4;
17777         else if (ill->ill_dhcpinit != 0)
17778             ill->ill_inputfn = ill_input_full_v4;
17779         else if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_RSVP].connf_head
17780             != NULL)
17781             ill->ill_inputfn = ill_input_full_v4;
17782         else if (ipst->ips_ip_cgtp_filter &&
17783             ipst->ips_ip_cgtp_filter_ops != NULL)
17784             ill->ill_inputfn = ill_input_full_v4;
17785         else
17786             ill->ill_inputfn = ill_input_short_v4;
17787     }
17788 }

17790 /*
17791  * Re-evaluate ill_inputfn for all the IPv4 ill's.
17792  * Used when RSVP and CGTP comes and goes.
17793  */
17794 void
17795 ill_set_inputfn_all(ip_stack_t *ipst)
17796 {
17797     ill_walk_context_t ctx;
17798     ill_t *ill;

17800     rw_enter(&ipst->ips_ill_g_lock, RW_READER);
17801     ill = ILL_START_WALK_V4(&ctx, ipst);
17802     for (; ill != NULL; ill = ill_next(&ctx, ill))
17803         ill_set_inputfn(ill);

17805     rw_exit(&ipst->ips_ill_g_lock);
17806 }

17808 /*
17809  * Set the physical address information for 'ill' to the contents of the
17810  * dl_notify_ind_t pointed to by 'mp'. Must be called as writer, and will be
17811  * asynchronous if 'ill' cannot immediately be quiesced -- in which case
17812  * EINPROGRESS will be returned.
17813  */
17814 int
17815 ill_set_phys_addr(ill_t *ill, mblk_t *mp)

```

```

17816 {
17817     ipsq_t *ipsq = ill->ill_phyint->phyint_ipsq;
17818     dl_notify_ind_t *dlindp = (dl_notify_ind_t *)mp->b_rptr;

17820     ASSERT(IAM_WRITER_IPSQ(ipsq));

17822     if (dlindp->dl_data != DL_IPV6_LINK_LAYER_ADDR &&
17823         dlindp->dl_data != DL_CURR_DEST_ADDR &&
17824         dlindp->dl_data != DL_CURR_PHYS_ADDR) {
17825         /* Changing DL_IPV6_TOKEN is not yet supported */
17826         return (0);
17827     }

17829     /*
17830     * We need to store up to two copies of 'mp' in 'ill'. Due to the
17831     * design of ipsq_pending_mp_add(), we can't pass them as separate
17832     * arguments to ill_set_phys_addr_tail(). Instead, chain them
17833     * together here, then pull 'em apart in ill_set_phys_addr_tail().
17834     */
17835     if ((mp = copyb(mp)) == NULL || (mp->b_cont = copyb(mp)) == NULL) {
17836         freemsg(mp);
17837         return (ENOMEM);
17838     }

17840     ipsq_current_start(ipsq, ill->ill_ipif, 0);

17842     /*
17843     * Since we'll only do a logical down, we can't rely on ipif_down
17844     * to turn on ILL_DOWN_IN_PROGRESS, or for the DL_BIND_ACK to reset
17845     * ILL_DOWN_IN_PROGRESS. We instead manage this separately for this
17846     * case, to quiesce ire's and nce's for ill_is_quiescent.
17847     */
17848     mutex_enter(&ill->ill_lock);
17849     ill->ill_state_flags |= ILL_DOWN_IN_PROGRESS;
17850     /* no more ire/nce addition allowed */
17851     mutex_exit(&ill->ill_lock);

17853     /*
17854     * If we can quiesce the ill, then set the address. If not, then
17855     * ill_set_phys_addr_tail() will be called from ipif_ill_refrele_tail().
17856     */
17857     ill_down_ipifs(ill, B_TRUE);
17858     mutex_enter(&ill->ill_lock);
17859     if (ill_is_quiescent(ill)) {
17860         /* call cannot fail since 'conn_t **' argument is NULL */
17861         (void) ipsq_pending_mp_add(NULL, ill->ill_ipif, ill->ill_rq,
17862             mp, ILL_DOWN);
17863         mutex_exit(&ill->ill_lock);
17864         return (EINPROGRESS);
17865     }
17866     mutex_exit(&ill->ill_lock);

17868     ill_set_phys_addr_tail(ipsq, ill->ill_rq, mp, NULL);
17869     return (0);
17870 }

17872 /*
17873 * When the allowed-ips link property is set on the datalink, IP receives a
17874 * DL_NOTE_ALLOWED_IPS notification that is processed in ill_set_allowed_ips()
17875 * to initialize the ill_allowed_ips[] array in the ill_t. This array is then
17876 * used to vet addresses passed to ip_sioclt_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
17931     *ill = q->q_ptr;
17932     mblk_t
17933     *addrmp2 = unlnkb(addrmp);
17934     dl_notify_ind_t *dlindp = (dl_notify_ind_t *)addrmp->b_rptr;
17935     uint_t
17936     addrlen, addroff;
17937     int
17938     status;

17936     ASSERT(IAM_WRITER_IPSQ(ipsq));

17938     addroff = dlindp->dl_addr_offset;
17939     addrlen = dlindp->dl_addr_length - ABS(ill->ill_sap_length);

17941     switch (dlindp->dl_data) {
17942     case DL_IPV6_LINK_LAYER_ADDR:
17943         ill_set_ndmp(ill, addrmp, addroff, addrlen);
17944         freemsg(addrmp2);
17945         break;

17947     case DL_CURR_DEST_ADDR:

```

```

17948     freemsg(ill->ill_dest_addr_mp);
17949     ill->ill_dest_addr = addrmp->b_rptr + addroff;
17950     ill->ill_dest_addr_mp = addrmp;
17951     if (ill->ill_isv6) {
17952         ill_setdesttoken(ill);
17953         ipif_setdestlinklocal(ill->ill_ipif);
17954     }
17955     freemsg(addrmp2);
17956     break;

17958 case DL_CURR_PHYS_ADDR:
17959     freemsg(ill->ill_phys_addr_mp);
17960     ill->ill_phys_addr = addrmp->b_rptr + addroff;
17961     ill->ill_phys_addr_mp = addrmp;
17962     ill->ill_phys_addr_length = addrlen;
17963     if (ill->ill_isv6)
17964         ill_set_ndmp(ill, addrmp2, addroff, addrlen);
17965     else
17966         freemsg(addrmp2);
17967     if (ill->ill_isv6) {
17968         ill_setdefaulttoken(ill);
17969         ipif_setlinklocal(ill->ill_ipif);
17970     }
17971     break;
17972 default:
17973     ASSERT(0);
17974 }

17976 /*
17977  * reset ILL_DOWN_IN_PROGRESS so that we can successfully add ires
17978  * as we bring the ipifs up again.
17979  */
17980 mutex_enter(&ill->ill_lock);
17981 ill->ill_state_flags &= ~ILL_DOWN_IN_PROGRESS;
17982 mutex_exit(&ill->ill_lock);
17983 /*
17984  * If there are ipifs to bring up, ill_up_ipifs() will return
17985  * EINPROGRESS, and ipsq_current_finish() will be called by
17986  * ip_rput_dlpi_writer() or arp_bringup_done() when the last ipif is
17987  * brought up.
17988  */
17989 status = ill_up_ipifs(ill, q, addrmp);
17990 if (status != EINPROGRESS)
17991     ipsq_current_finish(ipsq);
17992 }

17994 /*
17995  * Helper routine for setting the ill_nd_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_dlpi_send(ill, mp);

```

```

18081      /*
18082       * For IPv4, we would usually get EINPROGRESS because the ETHERTYPE ARP
18083       * streams have to be unbound. When all the DLPI exchanges are done,
18084       * ipsq_current_finish() will be called by arp_brngup_done(). The
18085       * remainder of ipif bringup via ill_up_ipifs() will also be done in
18086       * arp_brngup_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 SIOCLIFCONF for address family 'af' and store the result into a
18122   * dynamically-allocated 'lifcp' that will be 'bufsizep' bytes on success.
18123   */
18124  static int
18125  ip_lifconf_ioctl(ldi_handle_t lh, int af, struct lifconf *lifcp,
18126                  uint_t *bufsizep, cred_t *cr)
18127  {
18128      int err;
18129      struct lifnum lifn;

18131      bzero(&lifn, sizeof (lifn));
18132      lifn.lifn_family = af;
18133      lifn.lifn_flags = LIFC_UNDER_IPMP;

18135      if ((err = ip_ioctl(lh, SIOCLIFNUM, &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 SIOCLIFNUM and SIOCLIFCONF.
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, SIOCLIFCONF, lifcp, sizeof (*lifcp), cr);
18150      if (err != 0) {
18151          kmem_free(lifcp->lifc_buf, *bufsizep);
18152          return (err);
18153      }

18155      return (0);
18156  }

18158  /*
18159   * Helper for ip_interface_cleanup() that removes the loopback interface.
18160   */
18161  static void
18162  ip_loopback_removeif(ldi_handle_t lh, boolean_t isv6, cred_t *cr)
18163  {
18164      int err;
18165      struct lifreq lifr;

18167      bzero(&lifr, sizeof (lifr));
18168      (void) strcpy(lifr.lifr_name, ipif_loopback_name);

18170      /*
18171       * Attempt to remove the interface. It may legitimately not exist
18172       * (e.g. the zone administrator unplumbed it), so ignore ENXIO.
18173       */
18174      err = ip_ioctl(lh, SIOCLIFREMOVEIF, &lifr, sizeof (lifr), cr);
18175      if (err != 0 && err != ENXIO) {
18176          ip0dbg(("ip_loopback_removeif: IP%s SIOCLIFREMOVEIF failed: "
18177              "error %d\n", isv6 ? "v6" : "v4", err));
18178      }
18179  }

18181  /*
18182   * Helper for ip_interface_cleanup() that ensures no IP interfaces are in IPMP
18183   * groups and that IPMP data addresses are down. These conditions must be met
18184   * so that IPMP interfaces can be I_PUNLINK'd, as per ip_sioctl_plink_ipmp().
18185   */
18186  static void
18187  ip_ipmp_cleanup(ldi_handle_t lh, boolean_t isv6, cred_t *cr)
18188  {
18189      int af = isv6 ? AF_INET6 : AF_INET;
18190      int i, nifs;
18191      int err;
18192      uint_t bufsize;
18193      uint_t lifrsize = sizeof (struct lifreq);
18194      struct lifconf lifc;
18195      struct lifreq *lifrp;

18197      if ((err = ip_lifconf_ioctl(lh, af, &lifc, &bufsize, cr)) != 0) {
18198          cmn_err(CE_WARN, "ip_ipmp_cleanup: cannot get interface list "
18199              "(error %d); any IPMP interfaces cannot be shutdown", err);
18200          return;
18201      }

18203      nifs = lifc.lifc_len / lifrsize;
18204      for (lifrp = lifc.lifc_req, i = 0; i < nifs; i++, lifrp++) {
18205          err = ip_ioctl(lh, SIOCLIFFLAGS, lifrp, lifrsize, cr);
18206          if (err != 0) {
18207              cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot get "
18208                  "flags: error %d", lifrp->lifr_name, err);
18209              continue;
18210          }

```



```

18212     if (lifrp->lifr_flags & IFF_IPMP) {
18213         if ((lifrp->lifr_flags & (IFF_UP|IFF_DUPLICATE)) == 0)
18214             continue;

18216         lifrp->lifr_flags &= ~IFF_UP;
18217         err = ip_ioctl(lh, SIOCSLIFFLAGS, lifrp, lifrsize, cr);
18218         if (err != 0) {
18219             cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18220                 "bring down (error %d); IPMP interface may "
18221                 "not be shutdown", lifrp->lifr_name, err);
18222         }

18224         /*
18225          * Check if IFF_DUPLICATE is still set -- and if so,
18226          * reset the address to clear it.
18227          */
18228         err = ip_ioctl(lh, SIOCGLIFFLAGS, lifrp, lifrsize, cr);
18229         if (err != 0 || !(lifrp->lifr_flags & IFF_DUPLICATE))
18230             continue;

18232         err = ip_ioctl(lh, SIOCGLIFADDR, lifrp, lifrsize, cr);
18233         if (err != 0 || (err = ip_ioctl(lh, SIOCGLIFADDR,
18234             lifrp, lifrsize, cr)) != 0) {
18235             cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18236                 "reset DAD (error %d); IPMP interface may "
18237                 "not be shutdown", lifrp->lifr_name, err);
18238         }
18239         continue;
18240     }

18242     if (strchr(lifrp->lifr_name, IPIF_SEPARATOR_CHAR) == 0) {
18243         lifrp->lifr_groupname[0] = '\0';
18244         if ((err = ip_ioctl(lh, SIOCSLIFGROUPNAME, lifrp,
18245             lifrsize, cr)) != 0) {
18246             cmn_err(CE_WARN, "ip_ipmp_cleanup: %s: cannot "
18247                 "leave IPMP group (error %d); associated "
18248                 "IPMP interface may not be shutdown",
18249                 lifrp->lifr_name, err);
18250             continue;
18251         }
18252     }
18253 }

18255 kmem_free(lifc.lifc_buf, bufsize);
18256 }

18258 #define UDPDEV        "/devices/pseudo/udp@0:udp"
18259 #define UDP6DEV       "/devices/pseudo/udp6@0:udp6"

18261 /*
18262  * Remove the loopback interfaces and prep the IPMP interfaces to be torn down.
18263  * Non-loopback interfaces are either I_LINK'd or I_PLINK'd; the former go away
18264  * when the user-level processes in the zone are killed and the latter are
18265  * cleaned up by str_stack_shutdown().
18266  */
18267 void
18268 ip_interface_cleanup(ip_stack_t *ipst)
18269 {
18270     ldi_handle_t    lh;
18271     ldi_ident_t     li;
18272     cred_t          *cr;
18273     int             err;
18274     int             i;
18275     char            *devs[] = { UDP6DEV, UDPDEV };
18276     netstackid_t    stackid = ipst->ips_netstack->netstack_stackid;

```

```

18278     if ((err = ldi_ident_from_major(ddi_name_to_major("ip"), &li)) != 0) {
18279         cmn_err(CE_WARN, "ip_interface_cleanup: cannot get ldi ident:"
18280             " error %d", err);
18281         return;
18282     }

18284     cr = zone_get_kcred(netstackid_to_zoneid(stackid));
18285     ASSERT(cr != NULL);

18287     /*
18288      * NOTE: loop executes exactly twice and is hardcoded to know that the
18289      * first iteration is IPv6. (Unrolling yields repetitious code, hence
18290      * the loop.)
18291      */
18292     for (i = 0; i < 2; i++) {
18293         err = ldi_open_by_name(devs[i], FREAD|FWRITE, cr, &lh, li);
18294         if (err != 0) {
18295             cmn_err(CE_WARN, "ip_interface_cleanup: cannot open %s:"
18296                 " error %d", devs[i], err);
18297             continue;
18298         }

18300         ip_loopback_removeif(lh, i == 0, cr);
18301         ip_ipmp_cleanup(lh, i == 0, cr);

18303         (void) ldi_close(lh, FREAD|FWRITE, cr);
18304     }

18306     ldi_ident_release(li);
18307     crfree(cr);
18308 }

18310 /*
18311  * This needs to be in-sync with nic_event_t definition
18312  */
18313 static const char *
18314 ill_hook_event2str(nic_event_t event)
18315 {
18316     switch (event) {
18317     case NE_PLUMB:
18318         return ("PLUMB");
18319     case NE_UNPLUMB:
18320         return ("UNPLUMB");
18321     case NE_UP:
18322         return ("UP");
18323     case NE_DOWN:
18324         return ("DOWN");
18325     case NE_ADDRESS_CHANGE:
18326         return ("ADDRESS_CHANGE");
18327     case NE_LIF_UP:
18328         return ("LIF_UP");
18329     case NE_LIF_DOWN:
18330         return ("LIF_DOWN");
18331     case NE_IFINDEX_CHANGE:
18332         return ("IFINDEX_CHANGE");
18333     default:
18334         return ("UNKNOWN");
18335     }
18336 }

18338 void
18339 ill_nic_event_dispatch(ill_t *ill, lif_if_t lif, nic_event_t event,
18340     nic_event_data_t data, size_t datalen)
18341 {
18342     ip_stack_t        *ipst = ill->ill_ipst;
18343     hook_nic_event_int_t *info;

```

```

18344     const char      *str = NULL;

18346     /* create a new nic event info */
18347     if ((info = kmem_alloc(sizeof (*info), KM_NOSLEEP)) == NULL)
18348         goto fail;

18350     info->hnei_event.hne_nic = ill->ill_phyint->phyint_ifindex;
18351     info->hnei_event.hne_lif = lif;
18352     info->hnei_event.hne_event = event;
18353     info->hnei_event.hne_protocol = ill->ill_isv6 ?
18354         ipst->ips_ipv6_net_data : ipst->ips_ipv4_net_data;
18355     info->hnei_event.hne_data = NULL;
18356     info->hnei_event.hne_datalen = 0;
18357     info->hnei_stackid = ipst->ips_netstack->netstack_stackid;

18359     if (data != NULL && datalen != 0) {
18360         info->hnei_event.hne_data = kmem_alloc(datalen, KM_NOSLEEP);
18361         if (info->hnei_event.hne_data == NULL)
18362             goto fail;
18363         bcopy(data, info->hnei_event.hne_data, datalen);
18364         info->hnei_event.hne_datalen = datalen;
18365     }

18367     if (ddi_taskq_dispatch(eventq_queue_nic, ip_ne_queue_func, info,
18368         DDI_NOSLEEP) == DDI_SUCCESS)
18369         return;

18371 fail:
18372     if (info != NULL) {
18373         if (info->hnei_event.hne_data != NULL) {
18374             kmem_free(info->hnei_event.hne_data,
18375                 info->hnei_event.hne_datalen);
18376         }
18377         kmem_free(info, sizeof (hook_nic_event_t));
18378     }
18379     str = ill_hook_event2str(event);
18380     ip2dbg(("ill_nic_event_dispatch: could not dispatch %s nic event "
18381         "information for %s (ENOMEM)\n", str, ill->ill_name));
18382 }

18384 static int
18385 ipif_arp_up_done_tail(ipif_t *ipif, enum ip_resolver_action res_act)
18386 {
18387     int             err = 0;
18388     const in_addr_t *addr = NULL;
18389     nce_t           *nce = NULL;
18390     ill_t           *ill = ipif->ipif_ill;
18391     ill_t           *bound_ill;
18392     boolean_t       added_ipif = B_FALSE;
18393     uint16_t        state;
18394     uint16_t        flags;

18396     DTRACE_PROBE3(ipif_downup, char *, "ipif_arp_up_done_tail",
18397         ill_t *, ill, ipif_t *, ipif);
18398     if (ipif->ipif_lcl_addr != INADDR_ANY) {
18399         addr = &ipif->ipif_lcl_addr;
18400     }

18402     if ((ipif->ipif_flags & IPIF_UNNUMBERED) || addr == NULL) {
18403         if (res_act != Res_act_initial)
18404             return (EINVAL);
18405     }

18407     if (addr != NULL) {
18408         ipmp_illgrp_t    *illg = ill->ill_grp;

```

```

18410         /* add unicast nce for the local addr */

18412         if (IS_IPMP(ill)) {
18413             /*
18414              * If we're here via ipif_up(), then the ipif
18415              * won't be bound yet -- add it to the group,
18416              * which will bind it if possible. (We would
18417              * add it in ipif_up(), but deleting on failure
18418              * there is gruesome.) If we're here via
18419              * ipmp_ill_bind_ipif(), then the ipif has
18420              * already been added to the group and we
18421              * just need to use the binding.
18422              */
18423             if ((bound_ill = ipmp_ipif_bound_ill(ipif)) == NULL) {
18424                 bound_ill = ipmp_illgrp_add_ipif(illg, ipif);
18425                 if (bound_ill == NULL) {
18426                     /*
18427                      * We couldn't bind the ipif to an ill
18428                      * yet, so we have nothing to publish.
18429                      * Mark the address as ready and return.
18430                      */
18431                     ipif->ipif_addr_ready = 1;
18432                     return (0);
18433                 }
18434                 added_ipif = B_TRUE;
18435             }
18436         } else {
18437             bound_ill = ill;
18438         }

18440         flags = (NCE_F_MYADDR | NCE_F_PUBLISH | NCE_F_AUTHORITY |
18441             NCE_F_NONUD);
18442         /*
18443          * If this is an initial bring-up (or the ipif was never
18444          * completely brought up), do DAD. Otherwise, we're here
18445          * because IPMP has rebound an address to this ill: send
18446          * unsolicited advertisements (ARP announcements) to
18447          * inform others.
18448          */
18449         if (res_act == Res_act_initial || !ipif->ipif_addr_ready) {
18450             state = ND_UNCHANGED; /* compute in nce_add_common() */
18451         } else {
18452             state = ND_REACHABLE;
18453             flags |= NCE_F_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             }
18616         }
18617     }
18618 } else {
18619     ip0dbg(("arp_bringup_done: DL_BIND_REQ failed\n"));
18620 }
18621
18622 if ((err == 0) && (ill->ill_up_ipifs)) {
18623     err = ill_up_ipifs(ill, q, mpl);
18624     if (err == EINPROGRESS)
18625         return;
18626 }
18627
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 %p on ill %s\n", (void *)ipif,
18636         ipif->ipif_ill->ill_name));
18637     ill->ill_move_ipif = NULL;
18638     if (err == 0) {
18639         err = ipif_up(ipif, q, mpl);
18640         if (err == EINPROGRESS)
18641             return;
18642     }
18643 }
18644
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 }
18660
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));
18675
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 }
18720
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 }
18730
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         }
18869     }

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_refrel(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_ioctl_get_ifhwaddr and
19064 * ip_ioctl_get_lifhwaddr as the basic function of these two functions
19065 * is the same.
19066 *
19067 * The goal here is to find an IP interface that corresponds to the name
19068 * provided by the caller in the ifreq/lifreq structure held in the mblk_t
19069 * chain and to fill out a sockaddr/sockaddr_storage structure with the

```

```

19070 * mac address.
19071 *
19072 * The SIOCGIFHWADDR/SIOCGLIFHWADDR ioctl may return an error for a number
19073 * of different reasons:
19074 * ENXIO - the device name is not known to IP.
19075 * EADDRNOTAVAIL - the device has no hardware address. This is indicated
19076 * by ill_phys_addr not pointing to an actual address.
19077 * EPFNOSUPPORT - this will indicate that a request is being made for a
19078 * mac address that will not fit in the data structure supplier (struct
19079 * sockaddr).
19080 *
19081 */
19082 /* ARGSUSED */
19083 int
19084 ip_ioctl_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_ioctl_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_ioctl_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

```
*****
89772 Sat Aug 18 10:37:16 2012
new/usr/src/uts/common/inet/ip/ip_input.c
dccc: ips_ipcl_dccp_fanout
*****

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/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_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/squeue_impl.h>
104 #include <sys/squeue.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/ipgpc.h>

114 #include <sys/pattr.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_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     *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_sqmp != NULL) {
383         /* Better be called from ip_accept_tcp */
384         ASSERT(target_sqmp != NULL);

386         /* Found one packet to accept */
387         mp = iras.ira_target_sqmp;
388         iras.ira_target_sqmp = 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(nextthop)));
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     nextthop = htonl(INADDR_UNSPEC_GROUP);
547     ira->ira_flags |= IRAF_RSVP;
548 }

550     ill_input_short_v4(mp, ipha, &nextthop, 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 *nextthop_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     nextthop = *(ipaddr_t *)nextthop_arg;
569     ilb_stack_t *ilbs = ipst->ips_netstack->netstack_ilb;
570     uint_t      irr_flags;
571 #define rptra ((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 - rptra;

```

```

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, rptra, len, pkt_len,
596             IP_SIMPLE_HDR_LENGTH, ira);
597         if (mp == NULL)
598             return;
599         ipha = (ipha_t *)mp->b_rptra;
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_rptra;
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 nextthop.
642          */
643         if (orig_dst != ipha->ipha_dst)
644             nextthop = ipha->ipha_dst;
645         if (nextthop == INADDR_ANY) {
646             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInAddrErrors);
647             ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
648             freemsg(mp);
649             return;
650         }
651     }

653     if (ipst->ips_ip4_observe.he_interested) {
654         zoneid_t dzone;

```

```

656      /*
657       * On the inbound path the src zone will be unknown as
658       * this packet has come from the wire.
659       */
660      dzone = ip_get_zoneid_v4(nexthop, mp, ira, ALL_ZONES);
661      ipobs_hook(mp, IPOBS_HOOK_INBOUND, ALL_ZONES, dzone, ill, ipst);
662  }

664  /*
665   * If there is a good HW IP header checksum we clear the need
666   * look at the IP header checksum.
667   */
668  if ((DB_CKSUMFLAGS(mp) & HCK_IPV4_HDRCKSUM) &&
669      ILL_HCKSUM_CAPABLE(ill) && dohwcksum) {
670      /* Header checksum was ok. Clear the flag */
671      DB_CKSUMFLAGS(mp) &= ~HCK_IPV4_HDRCKSUM;
672      ira->ira_flags &= ~IRAF_VERIFY_IP_CKSUM;
673  }

675  /*
676   * Here we check to see if we machine is setup as
677   * L3 loadbalancer and if the incoming packet is for a VIP
678   *
679   * Check the following:
680   * - there is at least a rule
681   * - protocol of the packet is supported
682   */
683  if (ilb_has_rules(ilbs) && ILB_SUPP_L4(ipha->ipha_protocol)) {
684      ipaddr_t      lb_dst;
685      int           lb_ret;

687      /* For convenience, we pull up the mblk. */
688      if (mp->b_cont != NULL) {
689          if (pullupmsg(mp, -1) == 0) {
690              BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
691              ip_drop_input("ipIfStatsInDiscards - pullupmsg",
692                          mp, ill);
693              freemsg(mp);
694              return;
695          }
696          ipha = (ipha_t *)mp->b_rptr;
697      }

699      /*
700       * We just drop all fragments going to any VIP, at
701       * least for now....
702       */
703      if (ntohs(ipha->ipha_fragment_offset_and_flags) &
704          (IPH_MF | IPH_OFFSET)) {
705          if (!ilb_rule_match_vip_v4(ilbs, nexthop, NULL)) {
706              goto after_ilb;
707          }

709          ILB_KSTAT_UPDATE(ilbs, ip_frag_in, 1);
710          ILB_KSTAT_UPDATE(ilbs, ip_frag_dropped, 1);
711          BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
712          ip_drop_input("ILB fragment", mp, ill);
713          freemsg(mp);
714          return;
715      }
716      lb_ret = ilb_check_v4(ilbs, ill, mp, ipha, ipha->ipha_protocol,
717                          (uint8_t *)ipha + IPH_HDR_LENGTH(ipha), &lb_dst);

719      if (lb_ret == ILB_DROPPED) {
720          /* Is this the right counter to increase? */
721          BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);

```

```

722      ip_drop_input("ILB_DROPPED", mp, ill);
723      freemsg(mp);
724      return;
725  }
726  if (lb_ret == ILB_BALANCED) {
727      /* Set the dst to that of the chosen server */
728      nexthop = lb_dst;
729      DB_CKSUMFLAGS(mp) = 0;
730  }
731  }

733  after_ilb:
734      opt_len = ipha->ipha_version_and_hdr_length - IP_SIMPLE_HDR_VERSION;
735      ira->ira_ip_hdr_length = IP_SIMPLE_HDR_LENGTH;
736      if (opt_len != 0) {
737          int error = 0;

739          ira->ira_ip_hdr_length += (opt_len << 2);
740          ira->ira_flags |= IRAF_IPV4_OPTIONS;

742          /* IP Options present! Validate the length. */
743          mp = ip_check_optlen(mp, ipha, opt_len, pkt_len, ira);
744          if (mp == NULL)
745              return;

747          /* Might have changed */
748          ipha = (ipha_t *)mp->b_rptr;

750          /* Verify IP header checksum before parsing the options */
751          if ((ira->ira_flags & IRAF_VERIFY_IP_CKSUM) &&
752              ip_csum_hdr(ipha)) {
753              BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
754              ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
755              freemsg(mp);
756              return;
757          }
758          ira->ira_flags &= ~IRAF_VERIFY_IP_CKSUM;

760          /*
761           * Go off to ip_input_options which returns the next hop
762           * destination address, which may have been affected
763           * by source routing.
764           */
765          IP_STAT(ipst, ip_opt);

767          nexthop = ip_input_options(ipha, nexthop, mp, ira, &error);
768          if (error != 0) {
769              /*
770               * An ICMP error has been sent and the packet has
771               * been dropped.
772               */
773              return;
774          }
775      }

777      if (ill->ill_flags & ILLF_ROUTER)
778          irr_flags = IRR_ALLOCATE;
779      else
780          irr_flags = IRR_NONE;

782      /* Can not use route cache with TX since the labels can differ */
783      if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
784          if (CLASSSD(nexthop)) {
785              ire = ire_multicast(ill);
786          } else {
787              /* Match destination and label */

```

```

788         ire = ire_route_recursive_v4(nexthop, 0, NULL,
789         ALL_ZONES, ira->ira_tsl, MATCH_IRE_SECATTR,
790         irr_flags, ira->ira_xmit_hint, ipst, NULL, NULL,
791         NULL);
792     }
793     /* Update the route cache so we do the ire_refrele */
794     ASSERT(ire != NULL);
795     if (rtc->rtc_ire != NULL)
796         ire_refrele(rtc->rtc_ire);
797     rtc->rtc_ire = ire;
798     rtc->rtc_ipaddr = nexthop;
799 } else if (nexthop == rtc->rtc_ipaddr && rtc->rtc_ire != NULL) {
800     /* Use the route cache */
801     ire = rtc->rtc_ire;
802 } else {
803     /* Update the route cache */
804     if (CLASSD(nexthop)) {
805         ire = ire_multicast(ill);
806     } else {
807         /* Just match the destination */
808         ire = ire_route_recursive_dstonly_v4(nexthop, irr_flags,
809         ira->ira_xmit_hint, ipst);
810     }
811     ASSERT(ire != NULL);
812     if (rtc->rtc_ire != NULL)
813         ire_refrele(rtc->rtc_ire);
814     rtc->rtc_ire = ire;
815     rtc->rtc_ipaddr = nexthop;
816 }

818 ire->ire_ib_pkt_count++;

820 /*
821  * Based on ire_type and ire_flags call one of:
822  *   ire_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 rptra

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 refhold on ire_nce_cache as a result of this to avoid races
872     * where ire_nce_cache is deleted.
873     *
874     * This ensures that we don't forward if the interface is down since
875     * ipif_down removes all the nces.
876     */
877     mutex_enter(&ire->ire_lock);
878     nce = ire->ire_nce_cache;
879     if (nce == NULL) {
880         /* Not yet set up - try to set one up */
881         mutex_exit(&ire->ire_lock);
882         (void) ire_revalidate_nce(ire);
883         mutex_enter(&ire->ire_lock);
884         nce = ire->ire_nce_cache;
885         if (nce == NULL) {
886             mutex_exit(&ire->ire_lock);
887             /* The ire_dep_parent chain went bad, or no memory */
888             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
889             ip_drop_input("No ire_dep_parent", mp, ill);
890             freemsg(mp);
891             return;
892         }
893     }
894     nce_refhold(nce);
895     mutex_exit(&ire->ire_lock);

897     if (nce->nce_is_condemned) {
898         nce_t *nce1;

900         nce1 = ire_handle_condemned_nce(nce, ire, ipha, NULL, B_FALSE);
901         nce_refrele(nce);
902         if (nce1 == NULL) {
903             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
904             ip_drop_input("No nce", mp, ill);
905             freemsg(mp);
906             return;
907         }
908         nce = nce1;
909     }
910     dst_ill = nce->nce_ill;

912     /*
913     * Unless we are forwarding, drop the packet.
914     * We have to let source routed packets through if they go out
915     * the same interface i.e., they are 'ping -l' packets.
916     */
917     if (!(dst_ill->ill_flags & ILLF_ROUTER) &&
918         !(ip_source_routed(ipha, ipst) && dst_ill == ill)) {
919         if (ip_source_routed(ipha, ipst)) {

```

```

920         ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
921         icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
922         nce_refrele(nce);
923         return;
924     }
925     BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
926     ip_drop_input("ipIfStatsForwProhibits", mp, ill);
927     freemsg(mp);
928     nce_refrele(nce);
929     return;
930 }

932 if (ire->ire_zoneid != GLOBAL_ZONEID && ire->ire_zoneid != ALL_ZONES) {
933     ipaddr_t      dst = ipha->ipha_dst;

935     ire->ire_ib_pkt_count--;
936     /*
937      * Should only use IREs that are visible from the
938      * global zone for forwarding.
939      * Take a source route into account the same way as ip_input
940      * did.
941      */
942     if (ira->ira_flags & IRAF_IPV4_OPTIONS) {
943         int          error = 0;

945         dst = ip_input_options(ipha, dst, mp, ira, &error);
946         ASSERT(error == 0);      /* ip_input checked */
947     }
948     ire = ire_route_recursive_v4(dst, 0, NULL, GLOBAL_ZONEID,
949         ira->ira_tsl, MATCH_IRE_SECATTR,
950         (ill->ill_flags & ILLF_ROUTER) ? IRR_ALLOCATE : IRR_NONE,
951         ira->ira_xmit_hint, ipst, NULL, NULL, NULL);
952     ire->ire_ib_pkt_count++;
953     (*ire->ire_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_ip4firewall_forwarding,
987     ill, dst_ill, ipha, mp, mp, 0, ipst, error);

989     DTRACE_PROBE1(ip4_forwarding_end, mblk_t *, mp);

991     if (mp == NULL) {
992         nce_refrele(nce);
993         return;
994     }
995     /*
996      * Even if the destination was changed by the filter we use the
997      * forwarding decision that was made based on the address
998      * in ip_input.
999      */

1001     /* Might have changed */
1002     ipha = (ipha_t *)mp->b_rptr;
1003     ira->ira_pktlen = ntohs(ipha->ipha_length);
1004 }

1006 /* Packet is being forwarded. Turning off hwcksum flag. */
1007 DB_CKSUMFLAGS(mp) = 0;

1009 /*
1010  * Martian Address Filtering [RFC 1812, Section 5.3.7]
1011  * The loopback address check for both src and dst has already
1012  * been checked in ip_input
1013  * In the future one can envision adding RPF checks using number 3.
1014  * If we already checked the same source address we can skip this.
1015  */
1016 if (!(ira->ira_flags & IRAF_VERIFIED_SRC) ||
1017     src != ira->ira_verified_src) {
1018     switch (ipst->ips_src_check) {
1019         case 0:      break;
1020         case 2:
1021             if (ip_type_v4(src, ipst) == IRE_BROADCAST) {
1022                 BUMP_MIB(ill->ill_ip_mib,
1023                     ipIfStatsForwProhibits);
1024                 BUMP_MIB(ill->ill_ip_mib,
1025                     ipIfStatsInAddrErrors);
1026                 ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
1027                 freemsg(mp);
1028                 nce_refrele(nce);
1029                 return;
1030             }
1031             /* FALLTHRU */
1032         case 1:
1033             if (CLASSSD(src)) {
1034                 BUMP_MIB(ill->ill_ip_mib,
1035                     ipIfStatsForwProhibits);
1036                 BUMP_MIB(ill->ill_ip_mib,
1037                     ipIfStatsInAddrErrors);
1038                 ip_drop_input("ipIfStatsInAddrErrors", mp, ill);
1039                 freemsg(mp);
1040                 nce_refrele(nce);
1041                 return;
1042             }
1043             break;
1044     }
1045     /* Remember for next packet */
1046     ira->ira_flags |= IRAF_VERIFIED_SRC;
1047     ira->ira_verified_src = src;
1048 }
1049 }
1050 }
```



```

1052 /*
1053  * Check if packet is going out the same link on which it arrived.
1054  * Means we might need to send a redirect.
1055  */
1056 if (IS_ON_SAME_LAN(dst_ill, ill) && ipst->ips_ip_g_send_redirects) {
1057     ip_send_potential_redirect_v4(mp, ipha, ire, ira);
1058 }

1060 added_tx_len = 0;
1061 if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
1062     mblk_t *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->nce_ill;
1121     uint32_t pkt_len;
1122     uint32_t sum;
1123     iaflags_t iraflags = ira->ira_flags;
1124     ip_stack_t *ipst = ill->ill_ipst;
1125     iaflags_t ixaflags;

1127     if (ipha->ipha_ttl <= 1) {
1128         /* Perhaps the checksum was bad */
1129         if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1130             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1131             ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1132             freemsg(mp);
1133             return;
1134         }
1135         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1136         ip_drop_input("ICMP_TTL_EXCEEDED", mp, ill);
1137         icmp_time_exceeded(mp, ICMP_TTL_EXCEEDED, ira);
1138         return;
1139     }
1140     ipha->ipha_ttl--;
1141     /* Adjust the checksum to reflect the ttl decrement. */
1142     sum = (int)ipha->ipha_hdr_checksum + IP_HDR_CSUM_TTL_ADJUST;
1143     ipha->ipha_hdr_checksum = (uint16_t)(sum + (sum >> 16));

1145     /* Check if there are options to update */
1146     if (iraflags & IRAF_IPV4_OPTIONS) {
1147         ASSERT(ipha->ipha_version_and_hdr_length !=
1148             IP_SIMPLE_HDR_VERSION);
1149         ASSERT(!(iraflags & IRAF_VERIFY_IP_CKSUM));

1151         if (!ip_forward_options(mp, ipha, dst_ill, ira)) {
1152             /* ipIfStatsForwProhibits and ip_drop_input done */
1153             return;
1154         }

1156         ipha->ipha_hdr_checksum = 0;
1157         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
1158     }

1160     /* Initiate Write side IPPF processing before any fragmentation */
1161     if (IPP_ENABLED(IPP_FWD_OUT, ipst)) {
1162         /* ip_process translates an IS_UNDER_IPMP */
1163         mp = ip_process(IPP_FWD_OUT, mp, dst_ill, dst_ill);
1164         if (mp == NULL) {
1165             /* ip_drop_packet and MIB done */
1166             ip2dbg(("ire_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     ixaflags = IXAF_IS_IPV4 | IXAF_NO_DEV_FLOW_CTL;

1178     if (pkt_len > mtu) {
1179         /*
1180          * It needs fragging on its way out. If we haven't
1181          * verified the header checksum yet we do it now since
1182          * are going to put a surely good checksum in the
1183          * outgoing header, we have to make sure that it

```

```

1184         * was good coming in.
1185         */
1186         if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1187             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1188             ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1189             freemsg(mp);
1190             return;
1191         }
1192         if (ipha->ipha_fragment_offset_and_flags & IPH_DF_HTONS) {
1193             BUMP_MIB(dst_ill->ill_ip_mib, ipIfStatsOutFragFails);
1194             ip_drop_output("ipIfStatsOutFragFails", mp, dst_ill);
1195             if (iraflags & IRAF_SYSTEM_LABELED) {
1196                 /*
1197                  * Remove any CIPSO option added by
1198                  * tsol_ip_forward, and make sure we report
1199                  * a path MTU so that there
1200                  * is room to add such a CIPSO option for future
1201                  * packets.
1202                  */
1203                 mtu = tsol_pmtu_adjust(mp, mtu, added_tx_len,
1204                                     AF_INET);
1205             }
1206
1207             icmp_frag_needed(mp, mtu, ira);
1208             return;
1209         }
1210
1211         (void) ip_fragment_v4(mp, nce, ixaflags, pkt_len, mtu,
1212                             ira->ira_xmit_hint, GLOBAL_ZONEID, 0, ip_xmit, NULL);
1213         return;
1214     }
1215
1216     ASSERT(pkt_len == ntohs(((ipha_t *)mp->b_rptr->ipha_length));
1217     if (iraflags & IRAF_LOOPBACK_COPY) {
1218         /*
1219          * IXAF_NO_LOOP_ZONEID is not set hence 7th arg
1220          * is don't care
1221          */
1222         (void) ip_postfrag_loopcheck(mp, nce,
1223                                     ixaflags | IXAF_LOOPBACK_COPY,
1224                                     pkt_len, ira->ira_xmit_hint, GLOBAL_ZONEID, 0, NULL);
1225     } else {
1226         (void) ip_xmit(mp, nce, ixaflags, pkt_len, ira->ira_xmit_hint,
1227                     GLOBAL_ZONEID, 0, NULL);
1228     }
1229 }
1230
1231 /*
1232  * ire_recvf 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_recvf 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_recvf 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);
1318
1319     /* Tag for higher-level protocols */
1320     ira->ira_flags |= IRAF_BROADCAST;
1321
1322     /*
1323      * Whether local or directed broadcast forwarding: don't allow
1324      * for TCP.
1325      */
1326     if (ipha->ipha_protocol == IPPROTO_TCP) {
1327         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1328         ip_drop_input("ipIfStatsInDiscards", mp, ill);
1329         freemsg(mp);
1330         return;
1331     }
1332
1333     /*
1334      * So that we don't end up with dups, only one ill an IPMP group is
1335      * nominated to receive broadcast traffic.
1336      * If we have no cast_ill we are liberal and accept everything.
1337      */
1338     if (IS_UNDER_IPMP(ill)) {
1339         /* For an under ill_grp can change under lock */
1340         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1341         if (!ill->ill_nom_cast && ill->ill_grp != NULL &&
1342             ill->ill_grp->ig_cast_ill != NULL) {
1343             rw_exit(&ipst->ips_ill_g_lock);
1344             /* No MIB since this is normal operation */
1345             ip_drop_input("not nom_cast", mp, ill);
1346             freemsg(mp);
1347             return;
1348         }
1349         rw_exit(&ipst->ips_ill_g_lock);
1350
1351         ira->ira_ruifindex = ill_get_upper_ifindex(ill);
1352     }
1353
1354     /*
1355      * After reassembly and IPsec we will need to duplicate the
1356      * broadcast packet for all matching zones on the ill.
1357      */
1358     ira->ira_zoneid = ALL_ZONES;
1359
1360     /*
1361      * Check for directed broadcast i.e. ire->ire_ill is different than
1362      * the incoming ill.
1363      * The same broadcast address can be assigned to multiple interfaces
1364      * so have to check explicitly for that case by looking up the alt_ire
1365      */
1366     if (dst_ill == ill && !(ire->ire_flags & RTF_MULTIRT)) {
1367         /* Reassemble on the ill on which the packet arrived */
1368         ip_input_local_v4(ire, mp, ipha, ira);
1369         /* Restore */
1370         ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1371         return;
1372     }
1373
1374     /* Is there an IRE_BROADCAST on the incoming ill? */
1375     ipha_dst = ((ira->ira_flags & IRAF_DHCP_UNICAST) ? INADDR_BROADCAST :
1376         ipha->ipha_dst);
1377     alt_ire = ire_ftable_lookup_v4(ipha_dst, 0, 0, IRE_BROADCAST, ill,
1378         ALL_ZONES, ira->ira_tsl,
1379         MATCH_IRE_TYPE|MATCH_IRE_ILL|MATCH_IRE_SECATTR, 0, ipst, NULL);
1380     if (alt_ire != NULL) {
1381         /* Not a directed broadcast */

```

```

1382     /*
1383      * In the special case of multirouted broadcast
1384      * packets, we unconditionally need to "gateway"
1385      * them to the appropriate interface here so that reassembly
1386      * works. We know that the IRE_BROADCAST on cgt0 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     }

1452     /*
1453      * Clear the indication that this may have hardware
1454      * checksum as we are not using it for forwarding.
1455      */
1456     DB_CKSUMFLAGS(mp) = 0;

1458     /*
1459      * Adjust ttl to 2 (1+1 - the forward engine will decrement it by one.
1460      */
1461     ipha->ipha_ttl = ipst->ips_ip_broadcast_ttl + 1;
1462     ipha->ipha_hdr_checksum = 0;
1463     ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);

1465     /*
1466      * We use ip_forward_xmit to do any fragmentation.
1467      * and loopback copy on the outbound interface.
1468      *
1469      * Make it so that IXAF_LOOPBACK_COPY to be set on transmit side.
1470      */
1471     ira->ira_flags |= IRAF_LOOPBACK_COPY;

1473     nce = arp_nce_init(dst_ill, ipha->ipha_dst, IRE_BROADCAST);
1474     if (nce == NULL) {
1475         BUMP_MIB(dst_ill->ill_ip_mib, ipIfStatsOutDiscards);
1476         ip_drop_output("No nce", mp, dst_ill);
1477         freemsg(mp);
1478         goto done;
1479     }

1481     ip_forward_xmit_v4(nce, ill, mp, ipha, ira, dst_ill->ill_mc_mtu, 0);
1482     nce_refrele(nce);
1483 done:
1484     /* Restore */
1485     ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1486 }

1488 /*
1489  * ire_recvfn for IRE_MULTICAST.
1490  */
1491 void
1492 ire_recv_multicast_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1493     ip_recv_attr_t *ira)
1494 {
1495     ipha_t      *ipha = (ipha_t *)iph_arg;
1496     ill_t        *ill = ira->ira_ill;
1497     ip_stack_t   *ipst = ill->ill_ipst;

1499     ASSERT(ire->ire_ill == ira->ira_ill);

1501     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastPkts);
1502     UPDATE_MIB(ill->ill_ip_mib, ipIfStatsHCInMcastOctets, ira->ira_pktlen);

1504     /* RSVP hook */
1505     if (ira->ira_flags & IRAF_RSVP)
1506         goto forus;

1508     /* Tag for higher-level protocols */
1509     ira->ira_flags |= IRAF_MULTICAST;

1511     /*
1512      * So that we don't end up with dups, only one ill an IPMP group is
1513      * nominated to receive multicast traffic.

```

```

1514     * If we have no cast_ill we are liberal and accept everything.
1515     */
1516     if (IS_UNDER_IPMP(ill)) {
1517         ip_stack_t   *ipst = ill->ill_ipst;

1519         /* For an under ill_grp can change under lock */
1520         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1521         if (!ill->ill_nom_cast && ill->ill_grp != NULL &&
1522             ill->ill_grp->ig_cast_ill != NULL) {
1523             rw_exit(&ipst->ips_ill_g_lock);
1524             ip_drop_input("not on cast ill", mp, ill);
1525             freemsg(mp);
1526             return;
1527         }
1528         rw_exit(&ipst->ips_ill_g_lock);
1529     /*
1530      * We switch to the upper ill so that mrouter and hasmembers
1531      * can operate on upper here and in ip_input_multicast.
1532      */
1533     ill = ipmp_ill_hold_ipmp_ill(ill);
1534     if (ill != NULL) {
1535         ASSERT(ill != ira->ira_ill);
1536         ASSERT(ire->ire_ill == ira->ira_ill);
1537         ira->ira_ill = ill;
1538         ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;
1539     } else {
1540         ill = ira->ira_ill;
1541     }
1542 }

1544     /*
1545      * Check if we are a multicast router - send ip_mforward a copy of
1546      * the packet.
1547      * Due to mroute_decap tunnels we consider forwarding packets even if
1548      * mrouterd has not joined the allmulti group on this interface.
1549      */
1550     if (ipst->ips_ip_g_mrouter) {
1551         int retval;

1553         /*
1554          * Clear the indication that this may have hardware
1555          * checksum as we are not using it for forwarding.
1556          */
1557         DB_CKSUMFLAGS(mp) = 0;

1559         /*
1560          * ip_mforward helps us make these distinctions: If received
1561          * on tunnel and not IGMP, then drop.
1562          * If IGMP packet, then don't check membership
1563          * If received on a phyint and IGMP or PIM, then
1564          * don't check membership
1565          */
1566         retval = ip_mforward(mp, ira);
1567         /* ip_mforward updates mib variables if needed */

1569         switch (retval) {
1570             case 0:
1571                 /*
1572                  * pkt is okay and arrived on phyint.
1573                  */
1574                 /* If we are running as a multicast router
1575                  * we need to see all IGMP and/or PIM packets.
1576                  */
1577                 if ((ipha->ipha_protocol == IPPROTO_IGMP) ||
1578                     (ipha->ipha_protocol == IPPROTO_PIM)) {
1579                     goto forus;

```

```

1580         }
1581         break;
1582     case -1:
1583         /* pkt is mal-formed, toss it */
1584         freemsg(mp);
1585         goto done;
1586     case 1:
1587         /*
1588          * pkt is okay and arrived on a tunnel
1589          *
1590          * If we are running a multicast router
1591          * we need to see all igmp packets.
1592          */
1593         if (ipha->ipha_protocol == IPPROTO_IGMP) {
1594             goto forus;
1595         }
1596         ip_drop_input("Multicast on tunnel ignored", mp, ill);
1597         freemsg(mp);
1598         goto done;
1599     }
1600 }

1602 /*
1603  * Check if we have members on this ill. This is not necessary for
1604  * correctness because even if the NIC/GLD had a leaky filter, we
1605  * filter before passing to each conn_t.
1606  */
1607 if (!ill_hasmembers_v4(ill, ipha->ipha_dst)) {
1608     /*
1609      * Nobody interested
1610      *
1611      * This might just be caused by the fact that
1612      * multiple IP Multicast addresses map to the same
1613      * link layer multicast - no need to increment counter!
1614      */
1615     ip_drop_input("Multicast with no members", mp, ill);
1616     freemsg(mp);
1617     goto done;
1618 }
1619 forus:
1620 ip2dbg(("ire_rcv_multicast_v4: multicast for us: 0x%x\n",
1621     ntohs(ipha->ipha_dst)));

1623 /*
1624  * After reassembly and IPsec we will need to duplicate the
1625  * multicast packet for all matching zones on the ill.
1626  */
1627 ira->ira_zoneid = ALL_ZONES;

1629 /* Reassemble on the ill on which the packet arrived */
1630 ip_input_local_v4(ire, mp, ipha, ira);
1631 done:
1632 if (ill != ire->ire_ill) {
1633     ill_refrele(ill);
1634     ira->ira_ill = ire->ire_ill;
1635     ira->ira_ruifindex = ira->ira_ill->ill_phyint->phyint_ifindex;
1636 }
1637 }

1639 /*
1640  * ire_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;

1649     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInNoRoutes);
1650     ip_drop_input("Not forwarding out MULTIRT", mp, ill);
1651     freemsg(mp);
1652 }

1654 /*
1655  * ire_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;

1667     ira->ira_zoneid = GLOBAL_ZONEID;

1669     /* Switch to the lo0 ill for further processing */
1670     if (ire_ill != ill) {
1671         /*
1672          * Update ira_ill to be the ILL on which the IP address
1673          * is hosted.
1674          * No need to hold the ill since we have a hold on the ire
1675          */
1676         ASSERT(ira->ira_ill == ira->ira_rill);
1677         ira->ira_ill = ire_ill;

1679         ip_input_local_v4(ire, mp, ipha, ira);

1681         /* Restore */
1682         ASSERT(ira->ira_ill == ire_ill);
1683         ira->ira_ill = ill;
1684         return;
1685     }

1686     ip_input_local_v4(ire, mp, ipha, ira);
1687 }

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;

1700     /* Make a note for DAD that this address is in use */
1701     ire->ire_last_used_time = LBOLT_FASTPATH;

1703     /* Only target the IRE_LOCAL with the right zoneid. */
1704     ira->ira_zoneid = ire->ire_zoneid;

1706     /*
1707      * If the packet arrived on the wrong ill, we check that
1708      * this is ok.
1709      * If it is, then we ensure that we do the reassembly on
1710      * the ill on which the address is hosted. We keep ira_rill as
1711      * the one on which the packet arrived, so that IP_PKTINFO and

```

```

1712     * friends can report this.
1713     */
1714     if (ire_ill != ill) {
1715         ire_t *new_ire;

1717         new_ire = ip_check_multihome(&ipha->ipha_dst, ire, ill);
1718         if (new_ire == NULL) {
1719             /* Drop packet */
1720             BUMP_MIB(ill->ill_ip_mib, ipIfStatsForwProhibits);
1721             ip_drop_input("ipIfStatsInForwProhibits", mp, ill);
1722             freemsg(mp);
1723             return;
1724         }
1725         /*
1726          * Update ira_ill to be the ILL on which the IP address
1727          * is hosted. No need to hold the ill since we have a
1728          * hold on the ire. Note that we do the switch even if
1729          * new_ire == ire (for IPMP, ire would be the one corresponding
1730          * to the IPMP ill).
1731          */
1732         ASSERT(ira->ira_ill == ira->ira_rill);
1733         ira->ira_ill = new_ire->ire_ill;

1735         /* ira_ruifindex tracks the upper for ira_rill */
1736         if (IS_UNDER_IPMP(ill))
1737             ira->ira_ruifindex = ill_get_upper_ifindex(ill);

1739         ip_input_local_v4(new_ire, mp, ipha, ira);

1741         /* Restore */
1742         ASSERT(ira->ira_ill == new_ire->ire_ill);
1743         ira->ira_ill = ill;
1744         ira->ira_ruifindex = ill->ill_phyint->phyint_ifindex;

1746         if (new_ire != ire)
1747             ire_refrele(new_ire);
1748         return;
1749     }

1751     ip_input_local_v4(ire, mp, ipha, ira);
1752 }

1754 /*
1755  * Common function for packets arriving for the host. Handles
1756  * checksum verification, reassembly checks, etc.
1757  */
1758 static void
1759 ip_input_local_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
1760 {
1761     ill_t         *ill = ira->ira_ill;
1762     iaflags_t     iraflags = ira->ira_flags;

1764     /*
1765      * Verify IP header checksum. If the packet was AH or ESP then
1766      * this flag has already been cleared. Likewise if the packet
1767      * had a hardware checksum.
1768      */
1769     if ((iraflags & IRAF_VERIFY_IP_CKSUM) && ip_csum_hdr(ipha)) {
1770         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInCksumErrs);
1771         ip_drop_input("ipIfStatsInCksumErrs", mp, ill);
1772         freemsg(mp);
1773         return;
1774     }

1776     if (iraflags & IRAF_IPV4_OPTIONS) {
1777         if (!ip_input_local_options(mp, ipha, ira)) {

```

```

1778         /* Error has been sent and mp consumed */
1779         return;
1780     }
1781     /*
1782      * Some old hardware does partial checksum by including the
1783      * whole IP header, so the partial checksum value might have
1784      * become invalid if any option in the packet have been
1785      * updated. Always clear partial checksum flag here.
1786      */
1787     DB_CKSUMFLAGS(mp) &= ~HCK_PARTIALCKSUM;
1788 }

1790     /*
1791      * Is packet part of fragmented IP packet?
1792      * We compare against defined values in network byte order
1793      */
1794     if (ipha->ipha_fragment_offset_and_flags &
1795         (IPH_MF_HTONS | IPH_OFFSET_HTONS)) {
1796         /*
1797          * Make sure we have ira_l2src before we loose the original
1798          * mblk
1799          */
1800         if (!(ira->ira_flags & IRAF_L2SRC_SET))
1801             ip_setl2src(mp, ira, ira->ira_rill);

1803         mp = ip_input_fragment(mp, ipha, ira);
1804         if (mp == NULL)
1805             return;
1806         /* Completed reassembly */
1807         ipha = (ipha_t *)mp->b_rptr;
1808     }

1810     /*
1811      * For broadcast and multicast we need some extra work before
1812      * we call ip_fanout_v4(), since in the case of shared-IP zones
1813      * we need to pretend that a packet arrived for each zoneid.
1814      */
1815     if (iraflags & IRAF_MULTIBROADCAST) {
1816         if (iraflags & IRAF_BROADCAST)
1817             ip_input_broadcast_v4(ire, mp, ipha, ira);
1818         else
1819             ip_input_multicast_v4(ire, mp, ipha, ira);
1820         return;
1821     }
1822     ip_fanout_v4(mp, ipha, ira);
1823 }

1826 /*
1827  * Handle multiple zones which match the same broadcast address
1828  * and ill by delivering a packet to each of them.
1829  * Walk the bucket and look for different ire_zoneid but otherwise
1830  * the same IRE (same ill/addr/mask/type).
1831  * Note that ire_add() tracks IREs that are identical in all
1832  * fields (addr/mask/type/gw/ill/zoneid) within a single IRE by
1833  * increasing ire_identical_cnt. Thus we don't need to be concerned
1834  * about those.
1835  */
1836 static void
1837 ip_input_broadcast_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
1838 {
1839     ill_t         *ill = ira->ira_ill;
1840     ip_stack_t     *ipst = ill->ill_ipst;
1841     netstack_t     *ns = ipst->ips_netstack;
1842     irb_t          *irb;
1843     ire_t          *irel;

```

```

1844     mblk_t      *mpl;
1845     ipha_t      *ipha1;
1846     uint_t      ira_pktlen = ira->ira_pktlen;
1847     uint16_t     ira_ip_hdr_length = ira->ira_ip_hdr_length;

1849     irb = ire->ire_bucket;

1851     /*
1852     * If we don't have more than one shared-IP zone, or if
1853     * there can't be more than one IRE BROADCAST for this
1854     * IP address, then just set the zoneid and proceed.
1855     */
1856     if (ns->netstack_numzones == 1 || irb->irb_ire_cnt == 1) {
1857         ira->ira_zoneid = ire->ire_zoneid;

1859         ip_fanout_v4(mp, ipha, ira);
1860         return;
1861     }
1862     irb_refhold(irb);
1863     for (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         /*
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         if (IRE_IS_CONDEMNED(irel))
1878             continue;

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         ipha1 = (ipha_t *)mpl->b_rptr;
1889         ip_fanout_v4(mpl, ipha1, ira);
1890         /*
1891         * IPsec might have modified ira_pktlen and ira_ip_hdr_length
1892         * so we restore them for a potential next iteration
1893         */
1894         ira->ira_pktlen = ira_pktlen;
1895         ira->ira_ip_hdr_length = ira_ip_hdr_length;
1896     }
1897     irb_refrele(irb);
1898     /* Do the main ire */
1899     ira->ira_zoneid = ire->ire_zoneid;
1900     ip_fanout_v4(mp, ipha, ira);
1901 }

1903 /*
1904 * Handle multiple zones which want to receive the same multicast packets
1905 * on this ill by delivering a packet to each of them.
1906 *
1907 * Note that for packets delivered to transports we could instead do this
1908 * as part of the fanout code, but since we need to handle icmp_inbound
1909 * it is simpler to have multicast work the same as broadcast.

```

```

1910     *
1911     * The ip_fanout matching for multicast matches based on ilm independent of
1912     * zoneid since the zoneid restriction is applied when joining a multicast
1913     * group.
1914     */
1915     /* ARGSUSED */
1916     static void
1917     ip_input_multicast_v4(ire_t *ire, mblk_t *mp, ipha_t *ipha, ip_rcv_attr_t *ira)
1918     {
1919         ill_t      *ill = ira->ira_ill;
1920         iraflags_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      *ipha1;
1926         uint_t      ira_pktlen = ira->ira_pktlen;
1927         uint16_t     ira_ip_hdr_length = ira->ira_ip_hdr_length;

1929         /* ire_rcv_multicast has switched to the upper ill for IPMP */
1930         ASSERT(!IS_UNDER_IPMP(ill));

1932         /*
1933         * If we don't have more than one shared-IP zone, or if
1934         * there are no members in anything but the global zone,
1935         * then just set the zoneid and proceed.
1936         */
1937         if (ns->netstack_numzones == 1 ||
1938             !ill_hasmembers_otherzones_v4(ill, ipha->ipha_dst,
1939             GLOBAL_ZONEID)) {
1940             ira->ira_zoneid = GLOBAL_ZONEID;

1942             /* If sender didn't want this zone to receive it, drop */
1943             if ((iraflags & IRAF_NO_LOOP_ZONEID_SET) &&
1944                 ira->ira_no_loop_zoneid == ira->ira_zoneid) {
1945                 ip_drop_input("Multicast but wrong zoneid", mp, ill);
1946                 freemsg(mp);
1947                 return;
1948             }
1949             ip_fanout_v4(mp, ipha, ira);
1950             return;
1951         }

1953         /*
1954         * Here we loop over all zoneids that have members in the group
1955         * and deliver a packet to ip_fanout for each zoneid.
1956         *
1957         * First find any members in the lowest numeric zoneid by looking for
1958         * first zoneid larger than -1 (ALL_ZONES).
1959         * We terminate the loop when we receive -1 (ALL_ZONES).
1960         */
1961         zoneid = ill_hasmembers_nextzone_v4(ill, ipha->ipha_dst, ALL_ZONES);
1962         for (; zoneid != ALL_ZONES;
1963             zoneid = ill_hasmembers_nextzone_v4(ill, ipha->ipha_dst, zoneid)) {
1964             /*
1965             * Avoid an extra copymsg/freemsg by skipping global zone here
1966             * and doing that at the end.
1967             */
1968             if (zoneid == GLOBAL_ZONEID)
1969                 continue;

1971             ira->ira_zoneid = zoneid;

1973             /* If sender didn't want this zone to receive it, skip */
1974             if ((iraflags & IRAF_NO_LOOP_ZONEID_SET) &&
1975                 ira->ira_no_loop_zoneid == ira->ira_zoneid)

```

```

1976         continue;
1977
1978         mp1 = copymsg(mp);
1979         if (mp1 == NULL) {
1980             /* Failed to deliver to one zone */
1981             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1982             ip_drop_input("ipIfStatsInDiscards", mp, ill);
1983             continue;
1984         }
1985         ipha1 = (ipha_t *)mp1->b_rptr;
1986         ip_fanout_v4(mp1, ipha1, ira);
1987         /*
1988          * IPsec might have modified ira_pktlen and ira_ip_hdr_length
1989          * so we restore them for a potential next iteration
1990          */
1991         ira->ira_pktlen = ira_pktlen;
1992         ira->ira_ip_hdr_length = ira_ip_hdr_length;
1993     }
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  * Determine the zoneid and IRAF_TX_* flags if trusted extensions
2009  * is in use. Updates ira_zoneid and ira_flags as a result.
2010  */
2011 static void
2012 ip_fanout_tx_v4(mblk_t *mp, ipha_t *ipha, uint8_t protocol,
2013                uint_t ip_hdr_length, ip_rcv_attr_t *ira)
2014 {
2015     uint16_t    *up;
2016     uint16_t    lport;
2017     zoneid_t    zoneid;
2018
2019     ASSERT(ira->ira_flags & IRAF_SYSTEM_LABELED);
2020
2021     /*
2022      * If the packet is unlabeled we might allow read-down
2023      * for MAC_EXEMPT. Below we clear this if it is a multi-level
2024      * port (MLP).
2025      * Note that ira_tsl can be NULL here.
2026      */
2027     if (ira->ira_tsl != NULL && ira->ira_tsl->tsl_flags & TSLF_UNLABELED)
2028         ira->ira_flags |= IRAF_TX_MAC_EXEMPTABLE;
2029
2030     if (ira->ira_zoneid != ALL_ZONES)
2031         return;
2032
2033     ira->ira_flags |= IRAF_TX_SHARED_ADDR;
2034
2035     up = (uint16_t *)((uchar_t *)ipha + ip_hdr_length);
2036     switch (protocol) {
2037     case IPPROTO_TCP:
2038     case IPPROTO_SCTP:
2039     case IPPROTO_UDP:
2040         /* 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_recv_attr_t *ira)
2119 {
2120     uint_t      ulp_len;
2121     uint32_t     cksum;
2122     uint8_t      protocol = ira->ira_protocol;
2123     uint16_t     ip_hdr_length = ira->ira_ip_hdr_length;

2125 #define iphs      ((uint16_t *)ipha)

2127     switch (protocol) {
2128     case IPPROTO_TCP:
2129         ulp_len = ira->ira_pktlen - ip_hdr_length;

2131         /* Protocol and length */
2132         cksum = htons(ulp_len) + IP_TCP_CSUM_COMP;
2133         /* IP addresses */
2134         cksum += iphs[6] + iphs[7] + iphs[8] + iphs[9];
2135         break;

2137     case IPPROTO_UDP: {
2138         udpha_t      *udpha;

2140         udpha = (udpha_t *)((uchar_t *)ipha + ip_hdr_length);

2142         /* Protocol and length */
2143         cksum = udpha->uha_length + IP_UDP_CSUM_COMP;
2144         /* IP addresses */
2145         cksum += iphs[6] + iphs[7] + iphs[8] + iphs[9];
2146         break;
2147     }

2149     default:
2150         cksum = 0;
2151         break;
2152     }
2153 #undef iphs
2154     return (cksum);
2155 }

2158 /*
2159  * Software verification of the ULP checksums.
2160  * Returns B_TRUE if ok.
2161  * Increments statistics of failed.
2162  */
2163 static boolean_t
2164 ip_input_sw_cksum_v4(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
2165 {
2166     ip_stack_t      *ipst = ira->ira_ill->ill_ipst;
2167     uint32_t         cksum;
2168     uint8_t          protocol = ira->ira_protocol;
2169     uint16_t         ip_hdr_length = ira->ira_ip_hdr_length;

2171     IP_STAT(ipst, ip_in_sw_cksum);

2173     ASSERT(protocol == IPPROTO_TCP || protocol == IPPROTO_UDP);

```

```

2175     cksum = ip_input_cksum_pseudo_v4(ipha, ira);
2176     cksum = IP_CSUM(mp, ip_hdr_length, cksum);
2177     if (cksum == 0)
2178         return (B_TRUE);

2180     ip_input_cksum_err_v4(protocol, 0, ira->ira_ill);
2181     return (B_FALSE);
2182 }

2184 /*
2185  * Verify the ULP checksums.
2186  * Returns B_TRUE if ok, or if the ULP doesn't have a well-defined checksum
2187  * algorithm.
2188  * Increments statistics if failed.
2189  */
2190 static boolean_t
2191 ip_input_cksum_v4(iaflags_t iraflags, mblk_t *mp, ipha_t *ipha,
2192     ip_recv_attr_t *ira)
2193 {
2194     ill_t      *ill = ira->ira_rill;
2195     uint16_t    hck_flags;
2196     uint32_t    cksum;
2197     mblk_t      *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     mpl = mp->b_cont;
2288     if ((hck_flags & HCK_PARTIALCKSUM) &&
2289         (mpl == NULL || mpl->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 postpend data.

```

```

2306         /*
2307         IP_ADJCKSUM_PARTIAL(cksum_start, mp, mpl, len, adj);
2308         */
2309         /* One's complement subtract extraneous checksum
2310         */
2311         cksum += DB_CKSUM16(mp);
2312         if (adj >= cksum)
2313             cksum = ~(adj - cksum) & 0xFFFF;
2314         else
2315             cksum -= adj;
2316         cksum = (cksum & 0xFFFF) + ((int)cksum >> 16);
2317         cksum = (cksum & 0xFFFF) + ((int)cksum >> 16);
2318         if (!(~cksum & 0xFFFF))
2319             return (B_TRUE);
2321         ip_input_cksum_err_v4(protocol, hck_flags, ira->ira_ill);
2322         return (B_FALSE);
2323     }
2324     return (ip_input_sw_cksum_v4(mp, ipha, ira));
2325 }
2328 /*
2329 * Handle fanout of received packets.
2330 * Unicast packets that are looped back (from ire_send_local_v4) and packets
2331 * from the wire are differentiated by checking IRAF_VERIFY_ULP_CKSUM.
2332 *
2333 * IPQoS Notes
2334 * Before sending it to the client, invoke IPPF processing. Policy processing
2335 * takes place only if the callout_position, IPP_LOCAL_IN, is enabled.
2336 */
2337 void
2338 ip_fanout_v4(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
2339 {
2340     ill_t          *ill = ira->ira_ill;
2341     iaflags_t       iraflags = ira->ira_flags;
2342     ip_stack_t      *ipst = ill->ill_ipst;
2343     uint8_t         protocol = ipha->ipha_protocol;
2344     conn_t          *connp;
2345     #define rptr      ((uchar_t *)ipha)
2346     uint_t          ip_hdr_length;
2347     uint_t          min_ulp_header_length;
2348     int             offset;
2349     ssize_t         len;
2350     netstack_t      *ns = ipst->ips_netstack;
2351     ipsec_stack_t   *ipss = ns->netstack_ipsec;
2352     ill_t           *rill = ira->ira_rill;
2354     ASSERT(ira->ira_pktlen == ntohs(ipha->ipha_length));
2356     ip_hdr_length = ira->ira_ip_hdr_length;
2357     ira->ira_protocol = protocol;
2359     /*
2360     * Time for IPP once we've done reassembly and IPsec.
2361     * We skip this for loopback packets since we don't do IPQoS
2362     * on loopback.
2363     */
2364     if (IPP_ENABLED(IPP_LOCAL_IN, ipst) &&
2365         !(iraflags & IRAF_LOOPBACK) &&
2366         (protocol != IPPROTO_ESP || protocol != IPPROTO_AH)) {
2367         /*
2368         * Use the interface on which the packet arrived - not where
2369         * the IP address is hosted.
2370         */
2371         /* ip_process translates an IS_UNDER_IPMP */

```

```

2372         mp = ip_process(IPP_LOCAL_IN, mp, rill, ill);
2373         if (mp == NULL) {
2374             /* ip_drop_packet and MIB done */
2375             return;
2376         }
2377     }

2379     /* Determine the minimum required size of the upper-layer header */
2380     /* Need to do this for at least the set of ULPs that TX handles. */
2381     switch (protocol) {
2382     case IPPROTO_TCP:
2383         min_ulp_header_length = TCP_MIN_HEADER_LENGTH;
2384         break;
2385     case IPPROTO_SCTP:
2386         min_ulp_header_length = SCTP_COMMON_HDR_LENGTH;
2387         break;
2388     case IPPROTO_UDP:
2389         min_ulp_header_length = UDPH_SIZE;
2390         break;
2391     case IPPROTO_ICMP:
2392         min_ulp_header_length = ICMPH_SIZE;
2393         break;
2394     case IPPROTO_DCCP:
2395         min_ulp_header_length = DCCP_MIN_HEADER_LENGTH;
2396         break;
2397 #endif /* ! codereview */
2398     default:
2399         min_ulp_header_length = 0;
2400         break;
2401     }
2402     /* Make sure we have the min ULP header length */
2403     len = mp->b_wptr - rptr;
2404     if (len < ip_hdr_length + min_ulp_header_length) {
2405         if (ira->ira_pktlen < ip_hdr_length + min_ulp_header_length) {
2406             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
2407             ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
2408             freemsg(mp);
2409             return;
2410         }
2411         IP_STAT(ipst, ip_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;
2659     }

2660     if (connp->conn_incoming_ifindex != 0 &&
2661         connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2662         CONN_DEC_REF(connp);
2663         goto no_udp_match;
2664     }

2665     if (IPCL_IS_NONSTR(connp) ? connp->conn_flow_cntrl :
2666         !canputnext(connp->conn_rq)) {
2667         CONN_DEC_REF(connp);
2668         BUMP_MIB(ill->ill_ip_mib, udpIfStatsInOverflows);
2669         ip_drop_input("udpIfStatsInOverflows", mp, ill);
2670         freemsg(mp);
2671         return;
2672     }

2673     if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2674         (iraflags & IRAF_IPSEC_SECURE)) {
2675         mp = ipsec_check_inbound_policy(mp, connp,
2676             ipha, NULL, ira);
2677         if (mp == NULL) {
2678             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2679             /* Note that mp is NULL */
2680             ip_drop_input("ipIfStatsInDiscards", mp, ill);
2681             CONN_DEC_REF(connp);
2682             return;
2683         }
2684     }

2685     /*
2686     * Remove 0-spi if it's 0, or move everything behind
2687     * the UDP header over it and forward to ESP via
2688     * ip_fanout_v4().
2689     */
2690     if (connp->conn_udp->udp_nat_t_endpoint) {
2691         if (iraflags & IRAF_IPSEC_SECURE) {
2692             ip_drop_packet(mp, B_TRUE, ira->ira_ill,
2693                 DROPPER(ipss, ipds_esp_nat_t_ipsec),
2694                 &ipss->ipsec_dropper);
2695             CONN_DEC_REF(connp);
2696             return;
2697         }
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     /* First mblk contains IP+DCCP headers per above check */
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         /* Send the reset packet */
2742         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2743         dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2744         return;
2745     }

2747     if (connp->conn_incoming_ifindex != 0 &&
2748         connp->conn_incoming_ifindex != ira->ira_ruifindex) {
2749         cmn_err(CE_NOTE, "ip_input.c: ip_fanout_v4 ifindex probl
2750         /* Send the reset packet */
2751         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2752         dccp_xmit_listeners_reset(mp, ira, ipst, NULL);
2753         return;
2754     }

2756     if (CONN_INBOUND_POLICY_PRESENT(connp, ipss) ||
2757         (iraflags & IRAF_IPSEC_SECURE)) {
2758         mp = ipsec_check_inbound_policy(mp, connp,
2759             ipha, NULL, ira);
2760         if (mp == NULL) {
2761             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2762             /* Note that mp is NULL */
2763             ip_drop_input("ipIfStatsInDiscards", mp, ill);
2764             CONN_DEC_REF(connp);
2765             return;
2766         }
2767     }

```

```

2769      /* Found a client; up it goes */
2770      BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2771      ira->ira_ill = ira->ira_rill = NULL;

2773      /* XXX SOCK_RAW for DCCP? */

2775      if (iraflags & IRAF_TARGET_SQP) {
2776          cmn_err(CE_NOTE, "IRAF_TARGET_SQP");
2777      } else {
2778          SQUEUE_ENTER_ONE(connp->conn_sqp, mp, connp->conn_rcv,
2779                          connp, ira, ip_queue_flag, SQTAG_IP_DCCP_INPUT);
2780      }

2782      ira->ira_ill = ill;
2783      ira->ira_rill = rill;
2784      return;

2786 #endif /* ! codereview */
2787 default:
2788     break;
2789 }

2791 /*
2792  * Clear hardware checksumming flag as it is currently only
2793  * used by TCP and UDP.
2794  */
2795 DB_CKSUMFLAGS(mp) = 0;

2797 switch (protocol) {
2798 case IPPROTO_ICMP:
2799     /*
2800      * We need to accomodate icmp messages coming in clear
2801      * until we get everything secure from the wire. If
2802      * icmp_accept_clear_messages is zero we check with
2803      * the global policy and act accordingly. If it is
2804      * non-zero, we accept the message without any checks.
2805      * But *this does not mean* that this will be delivered
2806      * to RAW socket clients. By accepting we might send
2807      * replies back, change our MTU value etc.,
2808      * but delivery to the ULP/clients depends on their
2809      * policy dispositions.
2810      */
2811     if (ipst->ips_icmp_accept_clear_messages == 0) {
2812         mp = ipsec_check_global_policy(mp, NULL,
2813                                         ipha, NULL, ira, ns);
2814         if (mp == NULL)
2815             return;
2816     }

2818     /*
2819      * On a labeled system, we have to check whether the zone
2820      * itself is permitted to receive raw traffic.
2821      */
2822     if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
2823         if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
2824             BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
2825             ip_drop_input("tsol_can_accept_raw", mp, ill);
2826             freemsg(mp);
2827             return;
2828         }
2829     }

2831     /*
2832      * ICMP header checksum, including checksum field,
2833      * should be zero.

```

```

2834     /*
2835      * if (IP_CSUM(mp, ip_hdr_length, 0)) {
2836          BUMP_MIB(&ipst->ips_icmp_mib, icmpInCksumErrs);
2837          ip_drop_input("icmpInCksumErrs", mp, ill);
2838          freemsg(mp);
2839          return;
2840      }
2841      BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2842      mp = icmp_inbound_v4(mp, ira);
2843      if (mp == NULL) {
2844          /* No need to pass to RAW sockets */
2845          return;
2846      }
2847      break;

2849 case IPPROTO_IGMP:
2850     /*
2851      * If we are not willing to accept IGMP packets in clear,
2852      * then check with global policy.
2853      */
2854     if (ipst->ips_igmp_accept_clear_messages == 0) {
2855         mp = ipsec_check_global_policy(mp, NULL,
2856                                         ipha, NULL, ira, ns);
2857         if (mp == NULL)
2858             return;
2859     }
2860     if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2861         !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2862         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2863         ip_drop_input("ipIfStatsInDiscards", mp, ill);
2864         freemsg(mp);
2865         return;
2866     }
2867     /*
2868      * Validate checksum
2869      */
2870     if (IP_CSUM(mp, ip_hdr_length, 0)) {
2871         ++ipst->ips_igmpstat.igps_rcv_badsum;
2872         ip_drop_input("igps_rcv_badsum", mp, ill);
2873         freemsg(mp);
2874         return;
2875     }

2877     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2878     mp = igmp_input(mp, ira);
2879     if (mp == NULL) {
2880         /* Bad packet - discarded by igmp_input */
2881         return;
2882     }
2883     break;

2884 case IPPROTO_PIM:
2885     /*
2886      * If we are not willing to accept PIM packets in clear,
2887      * then check with global policy.
2888      */
2889     if (ipst->ips_pim_accept_clear_messages == 0) {
2890         mp = ipsec_check_global_policy(mp, NULL,
2891                                         ipha, NULL, ira, ns);
2892         if (mp == NULL)
2893             return;
2894     }
2895     if ((ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2896         !tsol_can_accept_raw(mp, ira, B_TRUE)) {
2897         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2898         ip_drop_input("ipIfStatsInDiscards", mp, ill);
2899         freemsg(mp);

```

```

2900         return;
2901     }
2902     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);

2904     /* Checksum is verified in pim_input */
2905     mp = pim_input(mp, ira);
2906     if (mp == NULL) {
2907         /* Bad packet - discarded by pim_input */
2908         return;
2909     }
2910     break;
2911 case IPPROTO_AH:
2912 case IPPROTO_ESP: {
2913     /*
2914      * Fast path for AH/ESP.
2915      */
2916     netstack_t *ns = ipst->ips_netstack;
2917     ipsec_stack_t *ipss = ns->netstack_ipsec;

2919     IP_STAT(ipst, ipsec_proto_ahesp);

2921     if (!ipsec_loaded(ipss)) {
2922         ip_proto_not_sup(mp, ira);
2923         return;
2924     }

2926     BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
2927     /* select inbound SA and have IPsec process the pkt */
2928     if (protocol == IPPROTO_ESP) {
2929         esp_h_t *esph;
2930         boolean_t esp_in_udp_sa;
2931         boolean_t esp_in_udp_packet;

2933         mp = ipsec_inbound_esp_sa(mp, ira, &esph);
2934         if (mp == NULL)
2935             return;

2937         ASSERT(esph != NULL);
2938         ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2939         ASSERT(ira->ira_ipsec_esp_sa != NULL);
2940         ASSERT(ira->ira_ipsec_esp_sa->ipsa_input_func != NULL);

2942         esp_in_udp_sa = ((ira->ira_ipsec_esp_sa->ipsa_flags &
2943             IPSA_F_NATT) != 0);
2944         esp_in_udp_packet =
2945             (ira->ira_flags & IRAF_ESP_UDP_PORTS) != 0;

2947         /*
2948          * The following is a fancy, but quick, way of saying:
2949          * ESP-in-UDP SA and Raw ESP packet --> drop
2950          * OR
2951          * ESP SA and ESP-in-UDP packet --> drop
2952          */
2953         if (esp_in_udp_sa != esp_in_udp_packet) {
2954             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
2955             ip_drop_packet(mp, B_TRUE, ira->ira_ill,
2956                 DROPPER(ipss, ipds_esp_no_sa),
2957                 &ipss->ipsec_dropper);
2958             return;
2959         }
2960         mp = ira->ira_ipsec_esp_sa->ipsa_input_func(mp, esph,
2961             ira);
2962     } else {
2963         ah_t *ah;

2965         mp = ipsec_inbound_ah_sa(mp, ira, &ah);

```

```

2966         if (mp == NULL)
2967             return;

2969         ASSERT(ah != NULL);
2970         ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
2971         ASSERT(ira->ira_ipsec_ah_sa != NULL);
2972         ASSERT(ira->ira_ipsec_ah_sa->ipsa_input_func != NULL);
2973         mp = ira->ira_ipsec_ah_sa->ipsa_input_func(mp, ah,
2974             ira);
2975     }

2977     if (mp == NULL) {
2978         /*
2979          * Either it failed or is pending. In the former case
2980          * ipIfStatsInDiscards was increased.
2981          */
2982         return;
2983     }
2984     /* we're done with IPsec processing, send it up */
2985     ip_input_post_ipsec(mp, ira);
2986     return;
2987 }
2988 case IPPROTO_ENCAP: {
2989     ipha_t             *inner_ipha;

2991     /*
2992      * Handle self-encapsulated packets (IP-in-IP where
2993      * the inner addresses == the outer addresses).
2994      */
2995     if ((uchar_t *)ipha + ip_hdr_length + sizeof (ipha_t) >
2996         mp->b_wptr) {
2997         if (ira->ira_pktlen <
2998             ip_hdr_length + sizeof (ipha_t)) {
2999             BUMP_MIB(ill->ill_ip_mib,
3000                 ipIfStatsInTruncatedPkts);
3001             ip_drop_input("ipIfStatsInTruncatedPkts",
3002                 mp, ill);
3003             freemsg(mp);
3004             return;
3005         }
3006         ipha = ip_pullup(mp, (uchar_t *)ipha + ip_hdr_length +
3007             sizeof (ipha_t) - mp->b_rptr, ira);
3008         if (ipha == NULL) {
3009             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3010             ip_drop_input("ipIfStatsInDiscards", mp, ill);
3011             freemsg(mp);
3012             return;
3013         }
3014     }
3015     inner_ipha = (ipha_t *)((uchar_t *)ipha + ip_hdr_length);
3016     /*
3017      * Check the sanity of the inner IP header.
3018      */
3019     if ((IPH_HDR_VERSION(inner_ipha) != IPV4_VERSION)) {
3020         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3021         ip_drop_input("ipIfStatsInDiscards", mp, ill);
3022         freemsg(mp);
3023         return;
3024     }
3025     if (IPH_HDR_LENGTH(inner_ipha) < sizeof (ipha_t)) {
3026         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3027         ip_drop_input("ipIfStatsInDiscards", mp, ill);
3028         freemsg(mp);
3029         return;
3030     }
3031     if (inner_ipha->ipha_src != ipha->ipha_src ||

```

```

3032         inner_ipha->ipha_dst != ipha->ipha_dst) {
3033             /* We fallthru to iptun fanout below */
3034             goto iptun;
3035         }
3036
3037     /*
3038     * Self-encapsulated tunnel packet. Remove
3039     * the outer IP header and fanout again.
3040     * We also need to make sure that the inner
3041     * header is pulled up until options.
3042     */
3043     mp->b_rptr = (uchar_t *)inner_ipha;
3044     ipha = inner_ipha;
3045     ip_hdr_length = IPH_HDR_LENGTH(ipha);
3046     if ((uchar_t *)ipha + ip_hdr_length > mp->b_wptr) {
3047         if (ira->ira_pktlen <
3048             (uchar_t *)ipha + ip_hdr_length - mp->b_rptr) {
3049             BUMP_MIB(ill->ill_ip_mib,
3050                 ipIfStatsInTruncatedPkts);
3051             ip_drop_input("ipIfStatsInTruncatedPkts",
3052                 mp, ill);
3053             freemsg(mp);
3054             return;
3055         }
3056         ipha = ip_pullup(mp,
3057             (uchar_t *)ipha + ip_hdr_length - mp->b_rptr, ira);
3058         if (ipha == NULL) {
3059             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3060             ip_drop_input("ipIfStatsInDiscards", mp, ill);
3061             freemsg(mp);
3062             return;
3063         }
3064     }
3065     if (ip_hdr_length > sizeof (ipha_t)) {
3066         /* We got options on the inner packet. */
3067         ipaddr_t      dst = ipha->ipha_dst;
3068         int            error = 0;
3069
3070         dst = ip_input_options(ipha, dst, mp, ira, &error);
3071         if (error != 0) {
3072             /*
3073              * An ICMP error has been sent and the packet
3074              * has been dropped.
3075              */
3076             return;
3077         }
3078         if (dst != ipha->ipha_dst) {
3079             /*
3080              * Someone put a source-route in
3081              * the inside header of a self-
3082              * encapsulated packet. Drop it
3083              * with extreme prejudice and let
3084              * the sender know.
3085              */
3086             ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
3087                 mp, ill);
3088             icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
3089                 ira);
3090             return;
3091         }
3092     }
3093     if (!(ira->ira_flags & IRAF_IPSEC_SECURE)) {
3094         /*
3095          * This means that somebody is sending
3096          * Self-encapsualted packets without AH/ESP.
3097          */

```

```

3098         * Send this packet to find a tunnel endpoint.
3099         * if I can't find one, an ICMP
3100         * PROTOCOL_UNREACHABLE will get sent.
3101         */
3102         protocol = ipha->ipha_protocol;
3103         ira->ira_protocol = protocol;
3104         goto iptun;
3105     }
3106
3107     /* Update based on removed IP header */
3108     ira->ira_ip_hdr_length = ip_hdr_length;
3109     ira->ira_pktlen = ntohs(ipha->ipha_length);
3110
3111     if (ira->ira_flags & IRAF_IPSEC_DECAPS) {
3112         /*
3113          * This packet is self-encapsulated multiple
3114          * times. We don't want to recurse infinitely.
3115          * To keep it simple, drop the packet.
3116          */
3117         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3118         ip_drop_input("ipIfStatsInDiscards", mp, ill);
3119         freemsg(mp);
3120         return;
3121     }
3122     ASSERT(ira->ira_flags & IRAF_IPSEC_SECURE);
3123     ira->ira_flags |= IRAF_IPSEC_DECAPS;
3124
3125     ip_input_post_ipsec(mp, ira);
3126     return;
3127 }
3128
3129 iptun: /* IPPROTO_ENCAPS that is not self-encapsulated */
3130 case IPPROTO_IPV6:
3131     /* iptun will verify trusted label */
3132     connp = ipcl_classify_v4(mp, protocol, ip_hdr_length,
3133         ira, ipst);
3134     if (connp != NULL) {
3135         BUMP_MIB(ill->ill_ip_mib, ipIfStatsHCInDelivers);
3136         ira->ira_ill = ira->ira_rill = NULL;
3137         (connp->conn_recv)(connp, mp, NULL, ira);
3138         CONN_DEC_REF(connp);
3139         ira->ira_ill = ill;
3140         ira->ira_rill = rill;
3141         return;
3142     }
3143     /* FALLTHRU */
3144 default:
3145     /*
3146      * On a labeled system, we have to check whether the zone
3147      * itself is permitted to receive raw traffic.
3148      */
3149     if (ira->ira_flags & IRAF_SYSTEM_LABELED) {
3150         if (!tsol_can_accept_raw(mp, ira, B_FALSE)) {
3151             BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3152             ip_drop_input("ipIfStatsInDiscards", mp, ill);
3153             freemsg(mp);
3154             return;
3155         }
3156     }
3157     break;
3158 }
3159
3160 /*
3161 * The above input functions may have returned the pulled up message.
3162 * So ipha need to be reinitialized.
3163 */

```



```
3164     ipha = (ipha_t *)mp->b_rptr;
3165     ira->ira_protocol = protocol = ipha->ipha_protocol;
3166     if (ipst->ips_ipcl_proto_fanout_v4[protocol].connf_head == NULL) {
3167         /*
3168          * No user-level listener for these packets packets.
3169          * Check for IPPROTO_ENCAP...
3170          */
3171         if (protocol == IPPROTO_ENCAP && ipst->ips_ip_g_mrouter) {
3172             /*
3173              * Check policy here,
3174              * THEN ship off to ip_mroute_decap().
3175              *
3176              * BTW, If I match a configured IP-in-IP
3177              * tunnel above, this path will not be reached, and
3178              * ip_mroute_decap will never be called.
3179              */
3180             mp = ipsec_check_global_policy(mp, connp,
3181             ipha, NULL, ira, ns);
3182             if (mp != NULL) {
3183                 ip_mroute_decap(mp, ira);
3184             } /* Else we already freed everything! */
3185         } else {
3186             ip_proto_not_sup(mp, ira);
3187         }
3188         return;
3189     }
3190
3191     /*
3192     * Handle fanout to raw sockets. There
3193     * can be more than one stream bound to a particular
3194     * protocol. When this is the case, each one gets a copy
3195     * of any incoming packets.
3196     */
3197     ASSERT(ira->ira_protocol == ipha->ipha_protocol);
3198     ip_fanout_proto_v4(mp, ipha, ira);
3199     return;
3200
3201 discard:
3202     BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
3203     ip_drop_input("ipIfStatsInDiscards", mp, ill);
3204     freemsg(mp);
3205 #undef rptr
3206 }
```

new/usr/src/uts/common/inet/ip/ip_output.c

1

```
*****
74252 Sat Aug 18 10:37:16 2012
new/usr/src/uts/common/inet/ip/ip_output.c
dccp: reset packet
*****
_____unchanged_portion_omitted_____

1606 /*
1607  * Calculate a checksum ignoring any hardware capabilities
1608  *
1609  * Returns B_FALSE if the packet was too short for the checksum. Caller
1610  * should free and do stats.
1611  */
1612 static boolean_t
1613 ip_output_sw_cksum_v4(mblk_t *mp, ipha_t *ipha, ip_xmit_attr_t *ixa)
1614 {
1615     ip_stack_t      *ipst = ixa->ixa_ipst;
1616     uint_t           pktlen = ixa->ixa_pktlen;
1617     uint16_t         *cksump;
1618     uint32_t         cksum;
1619     uint8_t          protocol = ixa->ixa_protocol;
1620     uint16_t         ip_hdr_length = ixa->ixa_ip_hdr_length;
1621     ipaddr_t         dst = ipha->ipha_dst;
1622     ipaddr_t         src = ipha->ipha_src;

1624     /* Just in case it contained garbage */
1625     DB_CKSUMFLAGS(mp) &= ~HCK_FLAGS;

1627     /*
1628      * Calculate ULP checksum
1629      */
1630     if (protocol == IPPROTO_TCP) {
1631         cksump = IPH_TCPH_CHECKSUMP(ipha, ip_hdr_length);
1632         cksum = IP_TCP_CSUM_COMP;
1633     } else if (protocol == IPPROTO_UDP) {
1634         cksump = IPH_UDPH_CHECKSUMP(ipha, ip_hdr_length);
1635         cksum = IP_UDP_CSUM_COMP;
1636     } else if (protocol == IPPROTO_SCTP) {
1637         sctp_hdr_t      *sctph;

1639         ASSERT(MBLKL(mp) >= (ip_hdr_length + sizeof(*sctph)));
1640         sctph = (sctp_hdr_t *) (mp->b_rptr + ip_hdr_length);
1641         /*
1642          * Zero out the checksum field to ensure proper
1643          * checksum calculation.
1644          */
1645         sctph->sh_chksum = 0;
1646 #ifdef DEBUG
1647         if (!skip_sctp_cksum)
1648             sctph->sh_chksum = sctp_cksum(mp, ip_hdr_length);
1649 #endif
1650         goto ip_hdr_cksum;
1651     } else if (protocol == IPPROTO_DCCP) {
1652         cksump = IPH_DCCPH_CHECKSUMP(ipha, ip_hdr_length);
1653         cksum = IP_DCCP_CSUM_COMP;
1654 #endif /* ! codereview */
1655     } else {
1656         goto ip_hdr_cksum;
1657     }

1659     /* ULP puts the checksum field is in the first mblk */
1660     ASSERT(((uchar_t *)cksump) + sizeof(uint16_t) <= mp->b_wptr);

1662     /*
1663      * We accumulate the pseudo header checksum in cksum.
1664      * This is pretty hairy code, so watch close. One
```

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_chksm = 0;
1739 #ifdef DEBUG
1740     if (!skip_sctp_cksum)
1741 #endif
1742         sctph->sh_chksm = sctp_cksum(mp, ip_hdr_length);
1743     goto ip_hdr_cksum;
1744 } else if (protocol == IPPROTO_DCCP) {
1745     cksum = IPH_DCCPH_CHECKSUM(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 is 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         error = 0;
2145
2146     /*
2147      * Check for IXAF_LOOPBACK_COPY - send a copy to ip as if the driver
2148      * had looped it back
2149      */
2150     if (iaflags & IXAF_LOOPBACK_COPY) {
2151         mblk_t      *mpl;
2152
2153         mpl = copymsg(mp);
2154         if (mpl == NULL) {
2155             /* Failed to deliver the loopback copy. */
2156             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2157             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
2158             error = ENOBUFS;
2159         } else {
2160             ip_postfrag_loopback(mpl, nce, iaflags, pkt_len,
2161                 nolzid);
2162         }
2163     }
2164
2165     /*
2166      * If TTL = 0 then only do the loopback to this host i.e. we are
2167      * done. We are also done if this was the
2168      * loopback interface since it is sufficient
2169      * to loopback one copy of a multicast packet.
2170      */
2171     if (iaflags & IXAF_IS_IPV4) {
2172         ipha_t *ipha = (ipha_t *)mp->b_rptr;
2173
2174         if (ipha->ipha_ttl == 0) {
2175             ip_drop_output("multicast ipha_ttl not sent to wire",
2176                 mp, ill);
2177             freemsg(mp);
2178             return (error);
2179         }
2180     } else {
2181         ip6_t      *ip6h = (ip6_t *)mp->b_rptr;
2182
2183         if (ip6h->ip6_hops == 0) {
2184             ip_drop_output("multicast ipha_ttl not sent to wire",
2185                 mp, ill);
2186             freemsg(mp);
2187             return (error);
2188         }
2189     }
2190     if (nce->nice_ill->ill_wq == NULL) {
2191         /* Loopback interface */
2192         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      *ncel;
2220     ill_t      *ill = nce->nice_ill;
2221     ill_t      *illl;
2222     ip_stack_t *ipst = ill->ill_ipst;
2223     int         error = 0;
2224     int         num_sent = 0;
2225     int         err;
2226     uint_t      ire_type;
2227     ipaddr_t    nexthop;
2228
2229     ASSERT(iaflags & IXAF_IS_IPV4);
2230
2231     /* Check for IXAF_LOOPBACK_COPY */
2232     if (iaflags & IXAF_LOOPBACK_COPY) {
2233         mblk_t *mpl;
2234
2235         mpl = copymsg(mp);
2236         if (mpl == NULL) {
2237             /* Failed to deliver the loopback copy. */
2238             BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2239             ip_drop_output("ipIfStatsOutDiscards", mp, ill);
2240             error = ENOBUFS;
2241         } else {
2242             ip_postfrag_loopback(mpl, nce, iaflags, pkt_len,
2243                 nolzid);
2244         }
2245     }
2246
2247     /*
2248      * Loop over RTF_MULTIRT for ipha_dst in the same bucket. Send
2249      * a copy to each one.
2250      * Use the nce (nexthop) and ipha_dst to find the ire.
2251      *
2252      * MULTIRT is not designed to work with shared-IP zones thus we don't
2253      * need to pass a zoneid or a label to the IRE lookup.
2254      */
2255     if (V4_PART_OF_V6(nce->nice_addr) == ipha->ipha_dst) {
2256         /* Broadcast and multicast case */
2257         ire = ire_fhtable_lookup_v4(ipha->ipha_dst, 0, 0, 0,
2258             NULL, ALL_ZONES, NULL, MATCH_IRE_DSTONLY, 0, ipst, NULL);

```

```

2259     } else {
2260         ipaddr_t v4addr = V4_PART_OF_V6(nce->nice_addr);

2262         /* Unicast case */
2263         ire = ire_fstable_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->nice_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->nice_common->nicec_flags & NCE_F_BCAST) {
2329         ire_type = IRE_BROADCAST;
2330         nexthop = irel->ire_gateway_addr;
2331     } else if (nce->nice_common->nicec_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 (ncel == 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_ip4firewall_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_ip4firewall_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 }

```

```

*****
90251 Sat Aug 18 10:37:17 2012
new/usr/src/uts/common/inet/ip/ipclassifier.c
dccb: conn_t
*****

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_aptun_fanout:    contains all IP tunnel connections
69 *     ipcl_globalhash_fanout: contains all connections
70 *     ipcl_dccb_conn_fanout: contains all DCCP connections in CONNECTED state
71 *     ipcl_dccb_bind_fanout: contains all DCCP connections in BOUND state
72 #endif /* ! codereview */
73 *
74 * The ipcl_globalhash_fanout is used for any walkers (like snmp and Clustering)
75 * which need to view all existing connections.
76 *
77 * All tables are protected by per-bucket locks. When both per-bucket lock and
78 * connection lock need to be held, the per-bucket lock should be acquired
79 * first, followed by the connection lock.
80 *
81 * All functions doing search in one of these tables increment a reference
82 * counter on the connection found (if any). This reference should be dropped
83 * when the caller has finished processing the connection.
84 *
85 *
86 * INTERFACES:
87 * =====
88 *
89 * Connection Lookup:
90 * -----
91 *
92 * conn_t *ipcl_classify_v4(mp, protocol, hdr_len, ira, ip_stack)
93 * conn_t *ipcl_classify_v6(mp, protocol, hdr_len, ira, ip_stack)
94 *
95 * Finds connection for an incoming IPv4 or IPv6 packet. Returns NULL if
96 * it can't find any associated connection. If the connection is found, its
97 * reference counter is incremented.
98 *
99 *     mp:      mblock, containing packet header. The full header should fit
100 *             into a single mblock. It should also contain at least full IP
101 *             and TCP or UDP header.
102 *
103 *     protocol: Either IPPROTO_TCP or IPPROTO_UDP.
104 *
105 *     hdr_len:  The size of IP header. It is used to find TCP or UDP header in
106 *             the packet.
107 *
108 *     ira->ira_zoneid: The zone in which the returned connection must be; the
109 *             zoneid corresponding to the ire_zoneid on the IRE located for
110 *             the packet's destination address.
111 *
112 *     ira->ira_flags: Contains the IRAF_TX_MAC_EXEMPTABLE and
113 *             IRAF_TX_SHARED_ADDR flags
114 *
115 * For TCP connections, the lookup order is as follows:
116 *     5-tuple {src, dst, protocol, local port, remote port}
117 *     lookup in ipcl_conn_fanout table.
118 *     3-tuple {dst, remote port, protocol} lookup in
119 *     ipcl_bind_fanout table.
120 *
121 * For UDP connections, a 5-tuple {src, dst, protocol, local port,
122 * remote port} lookup is done on ipcl_udp_fanout. Note that,
123 * these interfaces do not handle cases where a packets belongs
124 * to multiple UDP clients, which is handled in IP itself.
125 *
126 * If the destination IRE is ALL_ZONES (indicated by zoneid), then we must
127 * determine which actual zone gets the segment. This is used only in a

```

```

128 * labeled environment. The matching rules are:
129 *
130 * - If it's not a multilevel port, then the label on the packet selects
131 * the zone. Unlabeled packets are delivered to the global zone.
132 *
133 * - If it's a multilevel port, then only the zone registered to receive
134 * packets on that port matches.
135 *
136 * Also, in a labeled environment, packet labels need to be checked. For fully
137 * bound TCP connections, we can assume that the packet label was checked
138 * during connection establishment, and doesn't need to be checked on each
139 * packet. For others, though, we need to check for strict equality or, for
140 * multilevel ports, membership in the range or set. This part currently does
141 * a tnrrh lookup on each packet, but could be optimized to use cached results
142 * if that were necessary. (SCTP doesn't come through here, but if it did,
143 * we would apply the same rules as TCP.)
144 *
145 * An implication of the above is that fully-bound TCP sockets must always use
146 * distinct 4-tuples; they can't be discriminated by label alone.
147 *
148 * Note that we cannot trust labels on packets sent to fully-bound UDP sockets,
149 * as there's no connection set-up handshake and no shared state.
150 *
151 * Labels on looped-back packets within a single zone do not need to be
152 * checked, as all processes in the same zone have the same label.
153 *
154 * Finally, for unlabeled packets received by a labeled system, special rules
155 * apply. We consider only the MLP if there is one. Otherwise, we prefer a
156 * socket in the zone whose label matches the default label of the sender, if
157 * any. In any event, the receiving socket must have SO_MAC_EXEMPT set and the
158 * receiver's label must dominate the sender's default label.
159 *
160 * conn_t *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack);
161 * conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
162 * ip_stack);
163 *
164 * Lookup routine to find a exact match for {src, dst, local port,
165 * remote port} for TCP connections in ipcl_conn_fanout. The address and
166 * ports are read from the IP and TCP header respectively.
167 *
168 * conn_t *ipcl_lookup_listener_v4(lport, laddr, protocol,
169 * zoneid, ip_stack);
170 * conn_t *ipcl_lookup_listener_v6(lport, laddr, protocol, ifindex,
171 * zoneid, ip_stack);
172 *
173 * Lookup routine to find a listener with the tuple {lport, laddr,
174 * protocol} in the ipcl_bind_fanout table. For IPv6, an additional
175 * parameter interface index is also compared.
176 *
177 * void ipcl_walk(func, arg, ip_stack)
178 *
179 * Apply 'func' to every connection available. The 'func' is called as
180 * (*func)(connp, arg). The walk is non-atomic so connections may be
181 * created and destroyed during the walk. The CONN_CONDEMNED and
182 * CONN_INCIPIENT flags ensure that connections which are newly created
183 * or being destroyed are not selected by the walker.
184 *
185 * Table Updates
186 * -----
187 *
188 * int ipcl_conn_insert(connp);
189 * int ipcl_conn_insert_v4(connp);
190 * int ipcl_conn_insert_v6(connp);
191 *
192 * Insert 'connp' in the ipcl_conn_fanout.
193 * Arguments :

```

```

194 * connp conn_t to be inserted
195 *
196 * Return value :
197 * 0 if connp was inserted
198 * EADDRINUSE if the connection with the same tuple
199 * already exists.
200 *
201 * int ipcl_bind_insert(connp);
202 * int ipcl_bind_insert_v4(connp);
203 * int ipcl_bind_insert_v6(connp);
204 *
205 * Insert 'connp' in ipcl_bind_fanout.
206 * Arguments :
207 * connp conn_t to be inserted
208 *
209 *
210 * void ipcl_hash_remove(connp);
211 *
212 * Removes the 'connp' from the connection fanout table.
213 *
214 * Connection Creation/Destruction
215 * -----
216 *
217 * conn_t *ipcl_conn_create(type, sleep, netstack_t *)
218 *
219 * Creates a new conn based on the type flag, inserts it into
220 * globalhash table.
221 *
222 * type: This flag determines the type of conn_t which needs to be
223 * created i.e., which kmem_cache it comes from.
224 * IPCL_TCPCONN indicates a TCP connection
225 * IPCL_SCTPCONN indicates a SCTP connection
226 * IPCL_UDPCONN indicates a UDP conn_t.
227 * IPCL_RAWIPCONN indicates a RAWIP/ICMP conn_t.
228 * IPCL_RTSCONN indicates a RTS conn_t.
229 * IPCL_DCCPCONN indicates a DCCP conn_t.
230 *#endif /* ! codereview */
231 * IPCL_IPCCONN indicates all other connections.
232 *
233 * void ipcl_conn_destroy(connp)
234 *
235 * Destroys the connection state, removes it from the global
236 * connection hash table and frees its memory.
237 */
238
239 #include <sys/types.h>
240 #include <sys/stream.h>
241 #include <sys/stropts.h>
242 #include <sys/sysmacros.h>
243 #include <sys/strsubr.h>
244 #include <sys/strsun.h>
245 #define _SUN_TPI_VERSION 2
246 #include <sys/ddi.h>
247 #include <sys/cmn_err.h>
248 #include <sys/debug.h>
249
250 #include <sys/systm.h>
251 #include <sys/param.h>
252 #include <sys/kmem.h>
253 #include <sys/isa_defs.h>
254 #include <inet/common.h>
255 #include <netinet/ip6.h>
256 #include <netinet/icmp6.h>
257
258 #include <inet/ip.h>
259 #include <inet/ip_if.h>

```

```

260 #include <inet/ip_ire.h>
261 #include <inet/ip6.h>
262 #include <inet/ip_ndp.h>
263 #include <inet/ip_impl.h>
264 #include <inet/udp_impl.h>
265 #include <inet/dccp_impl.h>
266 #endif /* ! codereview */
267 #include <inet/sctp_ip.h>
268 #include <inet/sctp/sctp_impl.h>
269 #include <inet/rawip_impl.h>
270 #include <inet/rts_impl.h>
271 #include <inet/iptun/iptun_impl.h>

273 #include <sys/cpuvar.h>

275 #include <inet/ipclassifier.h>
276 #include <inet/tcp.h>
277 #include <inet/ipsec_impl.h>

279 #include <sys/tsol/tnet.h>
280 #include <sys/sockio.h>

282 /* Old value for compatibility. Setable in /etc/system */
283 uint_t tcp_conn_hash_size = 0;

285 /* New value. Zero means choose automatically. Setable in /etc/system */
286 uint_t ipcl_conn_hash_size = 0;
287 uint_t ipcl_conn_hash_memfactor = 8192;
288 uint_t ipcl_conn_hash_maxsize = 82500;

290 /* bind/udp fanout table size */
291 uint_t ipcl_bind_fanout_size = 512;
292 uint_t ipcl_udp_fanout_size = 16384;

294 /* Fanout table sizes for dccp */
295 uint_t ipcl_dccp_conn_fanout_size = 512;
296 uint_t ipcl_dccp_bind_fanout_size = 512;

298 #endif /* ! codereview */
299 /* Raw socket fanout size. Must be a power of 2. */
300 uint_t ipcl_raw_fanout_size = 256;

302 /*
303  * The IPCL_IPTUN_HASH() function works best with a prime table size. We
304  * expect that most large deployments would have hundreds of tunnels, and
305  * thousands in the extreme case.
306  */
307 uint_t ipcl_iptun_fanout_size = 6143;

309 /*
310  * Power of 2^N Primes useful for hashing for N of 0-28,
311  * these primes are the nearest prime <= 2^N - 2^(N-2).
312  */

314 #define P2Ps() {0, 0, 0, 5, 11, 23, 47, 89, 191, 383, 761, 1531, 3067, \
315                6143, 12281, 24571, 49139, 98299, 196597, 393209, \
316                786431, 1572853, 3145721, 6291449, 12582893, 25165813, \
317                50331599, 100663291, 201326557, 0}

319 /*
320  * wrapper structure to ensure that conn and what follows it (tcp_t, etc)
321  * are aligned on cache lines.
322  */
323 typedef union itc_s {
324     conn_t    itc_conn;
325     char      itcu_filler[CACHE_ALIGN(conn_s)];

```

```

326 } itc_t;

328 struct kmem_cache *tcp_conn_cache;
329 struct kmem_cache *ip_conn_cache;
330 extern struct kmem_cache *sctp_conn_cache;
331 struct kmem_cache *udp_conn_cache;
332 struct kmem_cache *rawip_conn_cache;
333 struct kmem_cache *rts_conn_cache;
334 struct kmem_cache *dccp_conn_cache;
335 #endif /* ! codereview */

337 extern void      tcp_timermp_free(tcp_t *);
338 extern mblk_t    *tcp_timermp_alloc(int);

340 static int      ip_conn_constructor(void *, void *, int);
341 static void     ip_conn_destructor(void *, void *);

343 static int      tcp_conn_constructor(void *, void *, int);
344 static void     tcp_conn_destructor(void *, void *);

346 static int      udp_conn_constructor(void *, void *, int);
347 static void     udp_conn_destructor(void *, void *);

349 static int      rawip_conn_constructor(void *, void *, int);
350 static void     rawip_conn_destructor(void *, void *);

352 static int      rts_conn_constructor(void *, void *, int);
353 static void     rts_conn_destructor(void *, void *);

355 static int      dccp_conn_constructor(void *, void *, int);
356 static void     dccp_conn_destructor(void *, void *);

358 #endif /* ! codereview */
359 /*
360  * Global (for all stack instances) init routine
361  */
362 void
363 ipcl_g_init(void)
364 {
365     ip_conn_cache = kmem_cache_create("ip_conn_cache",
366                                       sizeof(conn_t), CACHE_ALIGN_SIZE,
367                                       ip_conn_constructor, ip_conn_destructor,
368                                       NULL, NULL, NULL, 0);

370     tcp_conn_cache = kmem_cache_create("tcp_conn_cache",
371                                       sizeof(itc_t) + sizeof(tcp_t), CACHE_ALIGN_SIZE,
372                                       tcp_conn_constructor, tcp_conn_destructor,
373                                       tcp_conn_reclaim, NULL, NULL, 0);

375     udp_conn_cache = kmem_cache_create("udp_conn_cache",
376                                       sizeof(itc_t) + sizeof(udp_t), CACHE_ALIGN_SIZE,
377                                       udp_conn_constructor, udp_conn_destructor,
378                                       NULL, NULL, NULL, 0);

380     rawip_conn_cache = kmem_cache_create("rawip_conn_cache",
381                                       sizeof(itc_t) + sizeof(icmp_t), CACHE_ALIGN_SIZE,
382                                       rawip_conn_constructor, rawip_conn_destructor,
383                                       NULL, NULL, NULL, 0);

385     rts_conn_cache = kmem_cache_create("rts_conn_cache",
386                                       sizeof(itc_t) + sizeof(rts_t), CACHE_ALIGN_SIZE,
387                                       rts_conn_constructor, rts_conn_destructor,
388                                       NULL, NULL, NULL, 0);

390     /* XXX:DCCP reclaim */
391     dccp_conn_cache = kmem_cache_create("dccp_conn_cache",

```

```

392     sizeof (itc_t) + sizeof (dccp_t), CACHE_ALIGN_SIZE,
393     dccp_conn_constructor, dccp_conn_destructor,
394     NULL, NULL, NULL, 0);
395 #endif /* ! codereview */
396 }

398 /*
399  * ipclassifier intialization routine, sets up hash tables.
400  */
401 void
402 ipcl_init(ip_stack_t *ipst)
403 {
404     int i;
405     int sizes[] = P2Ps();

407     /*
408      * Calculate size of conn fanout table from /etc/system settings
409      */
410     if (ipcl_conn_hash_size != 0) {
411         ipst->ips_ipcl_conn_fanout_size = ipcl_conn_hash_size;
412     } else if (tcp_conn_hash_size != 0) {
413         ipst->ips_ipcl_conn_fanout_size = tcp_conn_hash_size;
414     } else {
415         extern pgcnt_t freemem;

417         ipst->ips_ipcl_conn_fanout_size =
418             (freemem * PAGESIZE) / ipcl_conn_hash_memfactor;

420         if (ipst->ips_ipcl_conn_fanout_size > ipcl_conn_hash_maxsize) {
421             ipst->ips_ipcl_conn_fanout_size =
422                 ipcl_conn_hash_maxsize;
423         }
424     }

426     for (i = 9; i < sizeof (sizes) / sizeof (*sizes) - 1; i++) {
427         if (sizes[i] >= ipst->ips_ipcl_conn_fanout_size) {
428             break;
429         }
430     }
431     if ((ipst->ips_ipcl_conn_fanout_size = sizes[i]) == 0) {
432         /* Out of range, use the 2^16 value */
433         ipst->ips_ipcl_conn_fanout_size = sizes[16];
434     }

436     /* Take values from /etc/system */
437     ipst->ips_ipcl_bind_fanout_size = ipcl_bind_fanout_size;
438     ipst->ips_ipcl_dccp_conn_fanout_size = ipcl_dccp_conn_fanout_size;
439     ipst->ips_ipcl_dccp_bind_fanout_size = ipcl_dccp_bind_fanout_size;
440 #endif /* ! codereview */
441     ipst->ips_ipcl_udp_fanout_size = ipcl_udp_fanout_size;
442     ipst->ips_ipcl_raw_fanout_size = ipcl_raw_fanout_size;
443     ipst->ips_ipcl_iptun_fanout_size = ipcl_iptun_fanout_size;

445     ASSERT(ipst->ips_ipcl_conn_fanout == NULL);

447     ipst->ips_ipcl_conn_fanout = kmem_zalloc(
448         ipst->ips_ipcl_conn_fanout_size * sizeof (connf_t), KM_SLEEP);

450     for (i = 0; i < ipst->ips_ipcl_conn_fanout_size; i++) {
451         mutex_init(&ipst->ips_ipcl_conn_fanout[i].connf_lock, NULL,
452             MUTEX_DEFAULT, NULL);
453     }

455     ipst->ips_ipcl_bind_fanout = kmem_zalloc(
456         ipst->ips_ipcl_bind_fanout_size * sizeof (connf_t), KM_SLEEP);

```

```

458     for (i = 0; i < ipst->ips_ipcl_bind_fanout_size; i++) {
459         mutex_init(&ipst->ips_ipcl_bind_fanout[i].connf_lock, NULL,
460             MUTEX_DEFAULT, NULL);
461     }

463     ipst->ips_ipcl_proto_fanout_v4 = 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_v4[i].connf_lock, NULL,
467             MUTEX_DEFAULT, NULL);
468     }

470     ipst->ips_ipcl_proto_fanout_v6 = kmem_zalloc(IPPROTO_MAX *
471         sizeof (connf_t), KM_SLEEP);
472     for (i = 0; i < IPPROTO_MAX; i++) {
473         mutex_init(&ipst->ips_ipcl_proto_fanout_v6[i].connf_lock, NULL,
474             MUTEX_DEFAULT, NULL);
475     }

477     ipst->ips_rts_clients = kmem_zalloc(sizeof (connf_t), KM_SLEEP);
478     mutex_init(&ipst->ips_rts_clients->connf_lock,
479         NULL, MUTEX_DEFAULT, NULL);

481     ipst->ips_ipcl_udp_fanout = kmem_zalloc(
482         ipst->ips_ipcl_udp_fanout_size * sizeof (connf_t), KM_SLEEP);
483     for (i = 0; i < ipst->ips_ipcl_udp_fanout_size; i++) {
484         mutex_init(&ipst->ips_ipcl_udp_fanout[i].connf_lock, NULL,
485             MUTEX_DEFAULT, NULL);
486     }

488     ipst->ips_ipcl_iptun_fanout = kmem_zalloc(
489         ipst->ips_ipcl_iptun_fanout_size * sizeof (connf_t), KM_SLEEP);
490     for (i = 0; i < ipst->ips_ipcl_iptun_fanout_size; i++) {
491         mutex_init(&ipst->ips_ipcl_iptun_fanout[i].connf_lock, NULL,
492             MUTEX_DEFAULT, NULL);
493     }

495     ipst->ips_ipcl_raw_fanout = kmem_zalloc(
496         ipst->ips_ipcl_raw_fanout_size * sizeof (connf_t), KM_SLEEP);
497     for (i = 0; i < ipst->ips_ipcl_raw_fanout_size; i++) {
498         mutex_init(&ipst->ips_ipcl_raw_fanout[i].connf_lock, NULL,
499             MUTEX_DEFAULT, NULL);
500     }

502     ipst->ips_ipcl_globalhash_fanout = kmem_zalloc(
503         sizeof (connf_t) * CONN_G_HASH_SIZE, KM_SLEEP);
504     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
505         mutex_init(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock,
506             NULL, MUTEX_DEFAULT, NULL);
507     }

509     ipst->ips_ipcl_dccp_conn_fanout = kmem_zalloc(
510         ipst->ips_ipcl_dccp_conn_fanout_size * sizeof (connf_t), KM_SLEEP);
511     for (i = 0; i < ipst->ips_ipcl_dccp_conn_fanout_size; i++) {
512         mutex_init(&ipst->ips_ipcl_dccp_conn_fanout[i].connf_lock, NULL,
513             MUTEX_DEFAULT, NULL);
514     }

516     ipst->ips_ipcl_dccp_bind_fanout = kmem_zalloc(
517         ipst->ips_ipcl_dccp_bind_fanout_size * sizeof (connf_t), KM_SLEEP);
518     for (i = 0; i < ipst->ips_ipcl_dccp_bind_fanout_size; i++) {
519         mutex_init(&ipst->ips_ipcl_dccp_bind_fanout[i].connf_lock, NULL,
520             MUTEX_DEFAULT, NULL);
521     }
522 #endif /* ! codereview */
523 }

```

```

525 void
526 ipcl_g_destroy(void)
527 {
528     kmem_cache_destroy(ip_conn_cache);
529     kmem_cache_destroy(tcp_conn_cache);
530     kmem_cache_destroy(udp_conn_cache);
531     kmem_cache_destroy(rawip_conn_cache);
532     kmem_cache_destroy(rts_conn_cache);
533     kmem_cache_destroy(dccp_conn_cache);
534 #endif /* ! codereview */
535 }

537 /*
538  * All user-level and kernel use of the stack must be gone
539  * by now.
540  */
541 void
542 ipcl_destroy(ip_stack_t *ipst)
543 {
544     int i;

546     for (i = 0; i < ipst->ips_ipcl_conn_fanout_size; i++) {
547         ASSERT(ipst->ips_ipcl_conn_fanout[i].connf_head == NULL);
548         mutex_destroy(&ipst->ips_ipcl_conn_fanout[i].connf_lock);
549     }
550     kmem_free(ipst->ips_ipcl_conn_fanout, ipst->ips_ipcl_conn_fanout_size *
551         sizeof (connf_t));
552     ipst->ips_ipcl_conn_fanout = NULL;

554     for (i = 0; i < ipst->ips_ipcl_bind_fanout_size; i++) {
555         ASSERT(ipst->ips_ipcl_bind_fanout[i].connf_head == NULL);
556         mutex_destroy(&ipst->ips_ipcl_bind_fanout[i].connf_lock);
557     }
558     kmem_free(ipst->ips_ipcl_bind_fanout, ipst->ips_ipcl_bind_fanout_size *
559         sizeof (connf_t));
560     ipst->ips_ipcl_bind_fanout = NULL;

562     for (i = 0; i < IPPROTO_MAX; i++) {
563         ASSERT(ipst->ips_ipcl_proto_fanout_v4[i].connf_head == NULL);
564         mutex_destroy(&ipst->ips_ipcl_proto_fanout_v4[i].connf_lock);
565     }
566     kmem_free(ipst->ips_ipcl_proto_fanout_v4,
567         IPPROTO_MAX * sizeof (connf_t));
568     ipst->ips_ipcl_proto_fanout_v4 = NULL;

570     for (i = 0; i < IPPROTO_MAX; i++) {
571         ASSERT(ipst->ips_ipcl_proto_fanout_v6[i].connf_head == NULL);
572         mutex_destroy(&ipst->ips_ipcl_proto_fanout_v6[i].connf_lock);
573     }
574     kmem_free(ipst->ips_ipcl_proto_fanout_v6,
575         IPPROTO_MAX * sizeof (connf_t));
576     ipst->ips_ipcl_proto_fanout_v6 = NULL;

578     for (i = 0; i < ipst->ips_ipcl_udp_fanout_size; i++) {
579         ASSERT(ipst->ips_ipcl_udp_fanout[i].connf_head == NULL);
580         mutex_destroy(&ipst->ips_ipcl_udp_fanout[i].connf_lock);
581     }
582     kmem_free(ipst->ips_ipcl_udp_fanout, ipst->ips_ipcl_udp_fanout_size *
583         sizeof (connf_t));
584     ipst->ips_ipcl_udp_fanout = NULL;

586     for (i = 0; i < ipst->ips_ipcl iptun_fanout_size; i++) {
587         ASSERT(ipst->ips_ipcl iptun_fanout[i].connf_head == NULL);
588         mutex_destroy(&ipst->ips_ipcl iptun_fanout[i].connf_lock);
589     }

```

```

590     kmem_free(ipst->ips_ipcl iptun_fanout,
591         ipst->ips_ipcl iptun_fanout_size * sizeof (connf_t));
592     ipst->ips_ipcl iptun_fanout = NULL;

594     for (i = 0; i < ipst->ips_ipcl_raw_fanout_size; i++) {
595         ASSERT(ipst->ips_ipcl_raw_fanout[i].connf_head == NULL);
596         mutex_destroy(&ipst->ips_ipcl_raw_fanout[i].connf_lock);
597     }
598     kmem_free(ipst->ips_ipcl_raw_fanout, ipst->ips_ipcl_raw_fanout_size *
599         sizeof (connf_t));
600     ipst->ips_ipcl_raw_fanout = NULL;

602     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
603         ASSERT(ipst->ips_ipcl_globalhash_fanout[i].connf_head == NULL);
604         mutex_destroy(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
605     }
606     kmem_free(ipst->ips_ipcl_globalhash_fanout,
607         sizeof (connf_t) * CONN_G_HASH_SIZE);
608     ipst->ips_ipcl_globalhash_fanout = NULL;

610     for (i = 0; i < ipst->ips_ipcl_dccp_conn_fanout_size; i++) {
611         ASSERT(ipst->ips_ipcl_dccp_conn_fanout[i].connf_head == NULL);
612         mutex_destroy(&ipst->ips_ipcl_dccp_conn_fanout[i].connf_lock);
613     }
614     kmem_free(ipst->ips_ipcl_dccp_conn_fanout,
615         ipst->ips_ipcl_dccp_conn_fanout_size * sizeof (connf_t));
616     ipst->ips_ipcl_dccp_conn_fanout = NULL;

618     for (i = 0; i < ipst->ips_ipcl_dccp_bind_fanout_size; i++) {
619         ASSERT(ipst->ips_ipcl_dccp_bind_fanout[i].connf_head == NULL);
620         mutex_destroy(&ipst->ips_ipcl_dccp_bind_fanout[i].connf_lock);
621     }
622     kmem_free(ipst->ips_ipcl_dccp_bind_fanout,
623         ipst->ips_ipcl_dccp_bind_fanout_size * sizeof (connf_t));
624     ipst->ips_ipcl_dccp_bind_fanout = NULL;

626 #endif /* ! codereview */
627     ASSERT(ipst->ips_rts_clients->connf_head == NULL);
628     mutex_destroy(&ipst->ips_rts_clients->connf_lock);
629     kmem_free(ipst->ips_rts_clients, sizeof (connf_t));
630     ipst->ips_rts_clients = NULL;
631 }

633 /*
634  * conn creation routine. initialize the conn, sets the reference
635  * and inserts it in the global hash table.
636  */
637 conn_t *
638 ipcl_conn_create(uint32_t type, int sleep, netstack_t *ns)
639 {
640     conn_t *connp;
641     struct kmem_cache *conn_cache;

643     switch (type) {
644     case IPCL_SCTPCONN:
645         if ((connp = kmem_cache_alloc(sctp_conn_cache, sleep)) == NULL)
646             return (NULL);
647         sctp_conn_init(connp);
648         netstack_hold(ns);
649         connp->conn_netstack = ns;
650         connp->conn_ixa->ixa_ipst = ns->netstack_ip;
651         connp->conn_ixa->ixa_conn_id = (long)connp;
652         ipcl_globalhash_insert(connp);
653         return (connp);

655     case IPCL_TCPCONN:

```

```

656         conn_cache = tcp_conn_cache;
657         break;

659     case IPCL_UDPCONN:
660         conn_cache = udp_conn_cache;
661         break;

663     case IPCL_RAWIPCONN:
664         conn_cache = rawip_conn_cache;
665         break;

667     case IPCL_RTSCONN:
668         conn_cache = rts_conn_cache;
669         break;

671     case IPCL_IPCCONN:
672         conn_cache = ip_conn_cache;
673         break;

675     case IPCL_DCCPCONN:
676         conn_cache = dccp_conn_cache;
677         break;

679 #endif /* ! codereview */
680 default:
681     connp = NULL;
682     ASSERT(0);
683 }

685 if ((connp = kmem_cache_alloc(conn_cache, sleep)) == NULL)
686     return (NULL);

688 connp->conn_ref = 1;
689 netstack_hold(ns);
690 connp->conn_netstack = ns;
691 connp->conn_ixa->ixa_ipst = ns->netstack_ip;
692 connp->conn_ixa->ixa_conn_id = (long)connp;
693 ipcl_globalhash_insert(connp);
694 return (connp);
695 }

697 void
698 ipcl_conn_destroy(conn_t *connp)
699 {
700     mblk_t *mp;
701     netstack_t *ns = connp->conn_netstack;

703     ASSERT(!MUTEX_HELD(&connp->conn_lock));
704     ASSERT(connp->conn_ref == 0);
705     ASSERT(connp->conn_ioctlref == 0);

707     DTRACE_PROBE1(conn_destroy, conn_t *, connp);

709     if (connp->conn_cred != NULL) {
710         crfree(connp->conn_cred);
711         connp->conn_cred = NULL;
712         /* ixa_cred done in ipcl_conn_cleanup below */
713     }

715     if (connp->conn_ht_iphc != NULL) {
716         kmem_free(connp->conn_ht_iphc, connp->conn_ht_iphc_allocated);
717         connp->conn_ht_iphc = NULL;
718         connp->conn_ht_iphc_allocated = 0;
719         connp->conn_ht_iphc_len = 0;
720         connp->conn_ht_ulp = NULL;
721         connp->conn_ht_ulp_len = 0;

```

```

722     }
723     ip_pkt_free(&connp->conn_xmit_ip);

725     ipcl_globalhash_remove(connp);

727     if (connp->conn_latch != NULL) {
728         IPLATCH_REFRELE(connp->conn_latch);
729         connp->conn_latch = NULL;
730     }
731     if (connp->conn_latch_in_policy != NULL) {
732         IPPOL_REFRELE(connp->conn_latch_in_policy);
733         connp->conn_latch_in_policy = NULL;
734     }
735     if (connp->conn_latch_in_action != NULL) {
736         IPACT_REFRELE(connp->conn_latch_in_action);
737         connp->conn_latch_in_action = NULL;
738     }
739     if (connp->conn_policy != NULL) {
740         IPPH_REFRELE(connp->conn_policy, ns);
741         connp->conn_policy = NULL;
742     }

744     if (connp->conn_ipsec_opt_mp != NULL) {
745         freemsg(connp->conn_ipsec_opt_mp);
746         connp->conn_ipsec_opt_mp = NULL;
747     }

749     if (connp->conn_flags & IPCL_TCPCONN) {
750         tcp_t *tcp = connp->conn_tcp;

752         tcp_free(tcp);
753         mp = tcp->tcp_timer_cache;

755         tcp->tcp_tcps = NULL;

757         /*
758          * tcp_rsrv_mp can be NULL if tcp_get_conn() fails to allocate
759          * the mblk.
760          */
761         if (tcp->tcp_rsrv_mp != NULL) {
762             freeb(tcp->tcp_rsrv_mp);
763             tcp->tcp_rsrv_mp = NULL;
764             mutex_destroy(&tcp->tcp_rsrv_mp_lock);
765         }

767         ipcl_conn_cleanup(connp);
768         connp->conn_flags = IPCL_TCPCONN;
769         if (ns != NULL) {
770             ASSERT(tcp->tcp_tcps == NULL);
771             connp->conn_netstack = NULL;
772             connp->conn_ixa->ixa_ipst = NULL;
773             netstack_rele(ns);
774         }

776         bzero(tcp, sizeof (tcp_t));

778         tcp->tcp_timer_cache = mp;
779         tcp->tcp_connp = connp;
780         kmem_cache_free(tcp_conn_cache, connp);
781         return;
782     }

784     if (connp->conn_flags & IPCL_SCTPCONN) {
785         ASSERT(ns != NULL);
786         sctp_free(connp);
787         return;

```

```

788     }
790     if (connp->conn_flags & IPCL_DCCPCONN) {
791         dccp_t *dccp = connp->conn_dccp;
793         cmn_err(CE_NOTE, "ipclassifier: conn_flags DCCP cache_free");
795         dccp_free(dccp);
796         mp = dccp->dccp_timercache;
798         dccp->dccp_dccps = NULL;
800         ipcl_conn_cleanup(connp);
801         connp->conn_flags = IPCL_DCCPCONN;
802         if (ns != NULL) {
803             ASSERT(dccp->dccps == NULL);
804             connp->conn_netstack = NULL;
805             connp->conn_ixa->ixa_ipst = NULL;
806             netstack_rele(ns);
807         }
809         bzero(dccp, sizeof (dccp_t));
811         dccp->dccp_timercache = mp;
812         dccp->dccp_connp = connp;
813         kmem_cache_free(dccp_conn_cache, connp);
814         return;
815     }
817 #endif /* ! codereview */
818 ipcl_conn_cleanup(connp);
819 if (ns != NULL) {
820     connp->conn_netstack = NULL;
821     connp->conn_ixa->ixa_ipst = NULL;
822     netstack_rele(ns);
823 }
825 /* leave conn_priv aka conn_udp, conn_icmp, etc in place. */
826 if (connp->conn_flags & IPCL_UDPCONN) {
827     connp->conn_flags = IPCL_UDPCONN;
828     kmem_cache_free(udp_conn_cache, connp);
829 } else if (connp->conn_flags & IPCL_RAWIPCONN) {
830     connp->conn_flags = IPCL_RAWIPCONN;
831     connp->conn_proto = IPPROTO_ICMP;
832     connp->conn_ixa->ixa_protocol = connp->conn_proto;
833     kmem_cache_free(rawip_conn_cache, connp);
834 } else if (connp->conn_flags & IPCL_RTSCONN) {
835     connp->conn_flags = IPCL_RTSCONN;
836     kmem_cache_free(rts_conn_cache, connp);
837 } else {
838     connp->conn_flags = IPCL_IPCCONN;
839     ASSERT(connp->conn_flags & IPCL_IPCCONN);
840     ASSERT(connp->conn_priv == NULL);
841     kmem_cache_free(ip_conn_cache, connp);
842 }
843 }
845 /*
846  * Running in cluster mode - deregister listener information
847  */
848 static void
849 ipcl_conn_unlisten(conn_t *connp)
850 {
851     ASSERT((connp->conn_flags & IPCL_CL_LISTENER) != 0);
852     ASSERT(connp->conn_lport != 0);

```

```

854     if (cl_inet_unlisten != NULL) {
855         sa_family_t  addr_family;
856         uint8_t      *laddrp;
858         if (connp->conn_ipversion == IPV6_VERSION) {
859             addr_family = AF_INET6;
860             laddrp = (uint8_t *)&connp->conn_bound_addr_v6;
861         } else {
862             addr_family = AF_INET;
863             laddrp = (uint8_t *)&connp->conn_bound_addr_v4;
864         }
865         (*cl_inet_unlisten)(connp->conn_netstack->netstack_stackid,
866             IPPROTO_TCP, addr_family, laddrp, connp->conn_lport, NULL);
867     }
868     connp->conn_flags &= ~IPCL_CL_LISTENER;
869 }
871 /*
872  * We set the IPCL_REMOVED flag (instead of clearing the flag indicating
873  * which table the conn belonged to). So for debugging we can see which hash
874  * table this connection was in.
875  */
876 #define IPCL_HASH_REMOVE(connp) {
877     connf_t *connpf = (connp)->conn_fanout;
878     ASSERT(!MUTEX_HELD(&((connp)->conn_lock)));
879     if (connpf != NULL) {
880         mutex_enter(&connpf->connpf_lock);
881         if ((connp)->conn_next != NULL)
882             (connp)->conn_next->conn_prev =
883                 (connp)->conn_prev;
884         if ((connp)->conn_prev != NULL)
885             (connp)->conn_prev->conn_next =
886                 (connp)->conn_next;
887         else
888             connp->connpf_head = (connp)->conn_next;
889         (connp)->conn_fanout = NULL;
890         (connp)->conn_next = NULL;
891         (connp)->conn_prev = NULL;
892         (connp)->conn_flags |= IPCL_REMOVED;
893         if (((connp)->conn_flags & IPCL_CL_LISTENER) != 0)
894             ipcl_conn_unlisten((connp));
895         CONN_DEC_REF((connp));
896         mutex_exit(&connpf->connpf_lock);
897     }
898 }
900 void
901 ipcl_hash_remove(conn_t *connp)
902 {
903     uint8_t      protocol = connp->conn_proto;
905     IPCL_HASH_REMOVE(connp);
906     if (protocol == IPPROTO_RSVP)
907         ill_set_inputfn_all(connp->conn_netstack->netstack_ip);
908 }
910 /*
911  * The whole purpose of this function is allow removal of
912  * a conn_t from the connected hash for timewait reclaim.
913  * This is essentially a TW reclaim fastpath where timewait
914  * collector checks under fanout lock (so no one else can
915  * get access to the conn_t) that refcnt is 2 i.e. one for
916  * TCP and one for the classifier hash list. If ref count
917  * is indeed 2, we can just remove the conn under lock and
918  * avoid cleaning up the conn under squeue. This gives us
919  * improved performance.

```



```

920 */
921 void
922 ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp)
923 {
924     ASSERT(MUTEX_HELD(&connfp->connf_lock));
925     ASSERT(MUTEX_HELD(&connp->conn_lock));
926     ASSERT((connp->conn_flags & IPCL_CL_LISTENER) == 0);

928     if ((connp->conn_next != NULL) {
929         (connp->conn_next->conn_prev = (connp->conn_prev);
930     }
931     if ((connp->conn_prev != NULL) {
932         (connp->conn_prev->conn_next = (connp->conn_next);
933     } else {
934         connfp->connf_head = (connp->conn_next);
935     }
936     (connp->conn_fanout = NULL;
937     (connp->conn_next = NULL;
938     (connp->conn_prev = NULL;
939     (connp->conn_flags |= IPCL_REMOVED;
940     ASSERT((connp->conn_ref == 2);
941     (connp->conn_ref--;
942 }

944 #define IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp) {
945     ASSERT((connp->conn_fanout == NULL);
946     ASSERT((connp->conn_next == NULL);
947     ASSERT((connp->conn_prev == NULL);
948     if ((connfp->connf_head != NULL) {
949         (connfp->connf_head->conn_prev = (connp);
950         (connp->conn_next = (connfp->connf_head);
951     }
952     (connp->conn_fanout = (connfp);
953     (connfp->connf_head = (connp);
954     (connp->conn_flags |= (connp->conn_flags & ~IPCL_REMOVED) |
955         IPCL_CONNECTED;
956     CONN_INC_REF(connp);
957 }

959 #define IPCL_HASH_INSERT_CONNECTED(connfp, connp) {
960     IPCL_HASH_REMOVE((connp));
961     mutex_enter(&(connfp->connf_lock);
962     IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
963     mutex_exit(&(connfp->connf_lock);
964 }

966 #define IPCL_HASH_INSERT_BOUND(connfp, connp) {
967     conn_t *pconnp = NULL, *nconnp;
968     IPCL_HASH_REMOVE((connp));
969     mutex_enter(&(connfp->connf_lock);
970     nconnp = (connfp->connf_head;
971     while (nconnp != NULL &&
972         !_IPCL_V4_MATCH_ANY(nconnp->conn_laddr_v6)) {
973         pconnp = nconnp;
974         nconnp = nconnp->conn_next;
975     }
976     if (pconnp != NULL) {
977         pconnp->conn_next = (connp);
978         (connp->conn_prev = pconnp;
979     } else {
980         (connfp->connf_head = (connp);
981     }
982     if (nconnp != NULL) {
983         (connp->conn_next = nconnp;
984         nconnp->conn_prev = (connp);
985     }

```

```

986     (connp->conn_fanout = (connfp);
987     (connp->conn_flags = ((connp->conn_flags & ~IPCL_REMOVED) |
988         IPCL_BOUND;
989     CONN_INC_REF(connp);
990     mutex_exit(&(connfp->connf_lock);
991 }

993 #define IPCL_HASH_INSERT_WILDCARD(connfp, connp) {
994     conn_t **list, *prev, *next;
995     boolean_t isv4mapped =
996         IN6_IS_ADDR_V4MAPPED(&(connp->conn_laddr_v6);
997     IPCL_HASH_REMOVE((connp));
998     mutex_enter(&(connfp->connf_lock);
999     list = &(connfp->connf_head;
1000     prev = NULL;
1001     while ((next = *list) != NULL) {
1002         if (isv4mapped &&
1003             IN6_IS_ADDR_UNSPECIFIED(&next->conn_laddr_v6) &&
1004             connp->conn_zoneid == next->conn_zoneid) {
1005             (connp->conn_next = next;
1006             if (prev != NULL)
1007                 prev = next->conn_prev;
1008             next->conn_prev = (connp);
1009             break;
1010         }
1011         list = &next->conn_next;
1012         prev = next;
1013     }
1014     (connp->conn_prev = prev;
1015     *list = (connp);
1016     (connp->conn_fanout = (connfp);
1017     (connp->conn_flags = ((connp->conn_flags & ~IPCL_REMOVED) |
1018         IPCL_BOUND;
1019     CONN_INC_REF((connp));
1020     mutex_exit(&(connfp->connf_lock);
1021 }

1023 void
1024 ipcl_hash_insert_wildcard(connf_t *connfp, conn_t *connp)
1025 {
1026     IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1027 }

1029 /*
1030  * Because the classifier is used to classify inbound packets, the destination
1031  * address is meant to be our local tunnel address (tunnel source), and the
1032  * source the remote tunnel address (tunnel destination).
1033  *
1034  * Note that conn_proto can't be used for fanout since the upper protocol
1035  * can be both 41 and 4 when IPv6 and IPv4 are over the same tunnel.
1036  */
1037 conn_t *
1038 ipcl_iptun_classify_v4(ipaddr_t *src, ipaddr_t *dst, ip_stack_t *ipst)
1039 {
1040     connf_t *connfp;
1041     conn_t *connp;

1043     /* first look for IPv4 tunnel links */
1044     connfp = &ipst->ips_ipcl_iptun_fanout[IPCL_IPTUN_HASH(*dst, *src)];
1045     mutex_enter(&connfp->connf_lock);
1046     for (connp = connfp->connf_head; connp != NULL;
1047         connp = connp->conn_next) {
1048         if (IPCL_IPTUN_MATCH(connp, *dst, *src))
1049             break;
1050     }
1051     if (connp != NULL)

```

```

1052         goto done;
1054         mutex_exit(&connfp->connf_lock);

1056         /* We didn't find an IPv4 tunnel, try a 6to4 tunnel */
1057         connfp = &ipst->ips_ipcl iptun_fanout[IPCL_IPTUN_HASH(*dst,
1058             INADDR_ANY)];
1059         mutex_enter(&connfp->connf_lock);
1060         for (connp = connfp->connf_head; connp != NULL;
1061             connp = connp->conn_next) {
1062             if (IPCL_IPTUN_MATCH(connp, *dst, INADDR_ANY))
1063                 break;
1064         }
1065     done:
1066     if (connp != NULL)
1067         CONN_INC_REF(connp);
1068     mutex_exit(&connfp->connf_lock);
1069     return (connp);
1070 }

1072 conn_t *
1073 ipcl iptun_classify_v6(in6_addr_t *src, in6_addr_t *dst, ip_stack_t *ipst)
1074 {
1075     connf_t *connfp;
1076     conn_t *connp;

1078     /* Look for an IPv6 tunnel link */
1079     connfp = &ipst->ips_ipcl iptun_fanout[IPCL_IPTUN_HASH_V6(dst, src)];
1080     mutex_enter(&connfp->connf_lock);
1081     for (connp = connfp->connf_head; connp != NULL;
1082         connp = connp->conn_next) {
1083         if (IPCL_IPTUN_MATCH_V6(connp, dst, src)) {
1084             CONN_INC_REF(connp);
1085             break;
1086         }
1087     }
1088     mutex_exit(&connfp->connf_lock);
1089     return (connp);
1090 }

1092 /*
1093  * This function is used only for inserting SCTP raw socket now.
1094  * This may change later.
1095  *
1096  * Note that only one raw socket can be bound to a port. The param
1097  * lport is in network byte order.
1098  */
1099 static int
1100 ipcl_sctp_hash_insert(conn_t *connp, in_port_t lport)
1101 {
1102     connf_t *connfp;
1103     conn_t *oconnp;
1104     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;

1106     connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(ntohs(lport), ipst)];

1108     /* Check for existing raw socket already bound to the port. */
1109     mutex_enter(&connfp->connf_lock);
1110     for (oconnp = connfp->connf_head; oconnp != NULL;
1111         oconnp = oconnp->conn_next) {
1112         if (oconnp->conn_lport == lport &&
1113             oconnp->conn_zoneid == connp->conn_zoneid &&
1114             oconnp->conn_family == connp->conn_family &&
1115             ((IN6_IS_ADDR_UNSPECIFIED(&oconnp->conn_laddr_v6) ||
1116              IN6_IS_ADDR_UNSPECIFIED(&oconnp->conn_laddr_v6) ||
1117              IN6_IS_ADDR_V4MAPPED_ANY(&oconnp->conn_laddr_v6) ||

```

```

1118             IN6_IS_ADDR_V4MAPPED_ANY(&oconnp->conn_laddr_v6)) ||
1119             IN6_IS_ADDR_EQUAL(&oconnp->conn_laddr_v6,
1120                 &connp->conn_laddr_v6)) {
1121                 break;
1122             }
1123     }
1124     mutex_exit(&connfp->connf_lock);
1125     if (oconnp != NULL)
1126         return (EADDRNOTAVAIL);

1128     if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) ||
1129         IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1130         if (IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6) ||
1131             IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_laddr_v6)) {
1132             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1133         } else {
1134             IPCL_HASH_INSERT_BOUND(connfp, connp);
1135         }
1136     } else {
1137         IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1138     }
1139     return (0);
1140 }

1142 static int
1143 ipcl iptun_hash_insert(conn_t *connp, ip_stack_t *ipst)
1144 {
1145     connf_t *connfp;
1146     conn_t *tconnp;
1147     ipaddr_t laddr = connp->conn_laddr_v4;
1148     ipaddr_t faddr = connp->conn_faddr_v4;

1150     connfp = &ipst->ips_ipcl iptun_fanout[IPCL_IPTUN_HASH(laddr, faddr)];
1151     mutex_enter(&connfp->connf_lock);
1152     for (tconnp = connfp->connf_head; tconnp != NULL;
1153         tconnp = tconnp->conn_next) {
1154         if (IPCL_IPTUN_MATCH(tconnp, laddr, faddr)) {
1155             /* A tunnel is already bound to these addresses. */
1156             mutex_exit(&connfp->connf_lock);
1157             return (EADDRINUSE);
1158         }
1159     }
1160     IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1161     mutex_exit(&connfp->connf_lock);
1162     return (0);
1163 }

1165 static int
1166 ipcl iptun_hash_insert_v6(conn_t *connp, ip_stack_t *ipst)
1167 {
1168     connf_t *connfp;
1169     conn_t *tconnp;
1170     in6_addr_t *laddr = &connp->conn_laddr_v6;
1171     in6_addr_t *faddr = &connp->conn_faddr_v6;

1173     connfp = &ipst->ips_ipcl iptun_fanout[IPCL_IPTUN_HASH_V6(laddr, faddr)];
1174     mutex_enter(&connfp->connf_lock);
1175     for (tconnp = connfp->connf_head; tconnp != NULL;
1176         tconnp = tconnp->conn_next) {
1177         if (IPCL_IPTUN_MATCH_V6(tconnp, laddr, faddr)) {
1178             /* A tunnel is already bound to these addresses. */
1179             mutex_exit(&connfp->connf_lock);
1180             return (EADDRINUSE);
1181         }
1182     }
1183     IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);

```

```

1184     mutex_exit(&connfp->connf_lock);
1185     return (0);
1186 }

1188 /*
1189  * Check for a MAC exemption conflict on a labeled system. Note that for
1190  * protocols that use port numbers (UDP, TCP, SCTP), we do this check up in the
1191  * transport layer. This check is for binding all other protocols.
1192  *
1193  * Returns true if there's a conflict.
1194  */
1195 static boolean_t
1196 check_exempt_conflict_v4(conn_t *connp, ip_stack_t *ipst)
1197 {
1198     connf_t *connfp;
1199     conn_t *tconn;

1201     connfp = &ipst->ips_ipcl_proto_fanout_v4[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 specific addrs, ok */
1216         if (connp->conn_laddr_v4 != INADDR_ANY &&
1217             tconn->conn_laddr_v4 != INADDR_ANY &&
1218             connp->conn_laddr_v4 != tconn->conn_laddr_v4)
1219             continue;
1220         /* These two conflict; fail */
1221         break;
1222     }
1223     mutex_exit(&connfp->connf_lock);
1224     return (tconn != NULL);
1225 }

1227 static boolean_t
1228 check_exempt_conflict_v6(conn_t *connp, ip_stack_t *ipst)
1229 {
1230     connf_t *connfp;
1231     conn_t *tconn;

1233     connfp = &ipst->ips_ipcl_proto_fanout_v6[connp->conn_proto];
1234     mutex_enter(&connfp->connf_lock);
1235     for (tconn = connfp->connf_head; tconn != NULL;
1236          tconn = tconn->conn_next) {
1237         /* We don't allow v4 fallback for v6 raw socket */
1238         if (connp->conn_family != tconn->conn_family)
1239             continue;
1240         /* If neither is exempt, then there's no conflict */
1241         if ((connp->conn_mac_mode == CONN_MAC_DEFAULT) &&
1242             (tconn->conn_mac_mode == CONN_MAC_DEFAULT))
1243             continue;
1244         /* We are only concerned about sockets for a different zone */
1245         if (connp->conn_zoneid == tconn->conn_zoneid)
1246             continue;
1247         /* If both are bound to different addrs, ok */
1248         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6) &&
1249             !IN6_IS_ADDR_UNSPECIFIED(&tconn->conn_laddr_v6) &&

```

```

1250         !IN6_ADDR_EQUAL(&connp->conn_laddr_v6,
1251             &tconn->conn_laddr_v6))
1252             continue;
1253         /* These two conflict; fail */
1254         break;
1255     }
1256     mutex_exit(&connfp->connf_lock);
1257     return (tconn != NULL);
1258 }

1260 /*
1261  * (v4, v6) bind hash insertion routines
1262  * The caller has already setup the conn (conn_proto, conn_laddr_v6, conn_lport)
1263  */

1265 int
1266 ipcl_bind_insert(conn_t *connp)
1267 {
1268     if (connp->conn_ipversion == IPV6_VERSION)
1269         return (ipcl_bind_insert_v6(connp));
1270     else
1271         return (ipcl_bind_insert_v4(connp));
1272 }

1274 int
1275 ipcl_bind_insert_v4(conn_t *connp)
1276 {
1277     connf_t *connfp;
1278     int ret = 0;
1279     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;
1280     uint16_t lport = connp->conn_lport;
1281     uint8_t protocol = connp->conn_proto;

1283     if (IPCL_IS_IPTUN(connp))
1284         return (ipcl_iptun_hash_insert(connp, ipst));

1286     switch (protocol) {
1287     default:
1288         if (is_system_labeled() &&
1289             check_exempt_conflict_v4(connp, ipst))
1290             return (EADDRINUSE);
1291         /* FALLTHROUGH */
1292     case IPPROTO_UDP:
1293         if (protocol == IPPROTO_UDP) {
1294             connfp = &ipst->ips_ipcl_udp_fanout[
1295                 IPCL_UDP_HASH(lport, ipst)];
1296         } else {
1297             connfp = &ipst->ips_ipcl_proto_fanout_v4[protocol];
1298         }

1300         if (connp->conn_faddr_v4 != INADDR_ANY) {
1301             IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1302         } else if (connp->conn_laddr_v4 != INADDR_ANY) {
1303             IPCL_HASH_INSERT_BOUND(connfp, connp);
1304         } else {
1305             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1306         }
1307         if (protocol == IPPROTO_RSVP)
1308             ill_set_inputfn_all(ipst);
1309         break;

1311     case IPPROTO_TCP:
1312         /* Insert it in the Bind Hash */
1313         ASSERT(connp->conn_zoneid != ALL_ZONES);
1314         connfp = &ipst->ips_ipcl_bind_fanout[
1315             IPCL_BIND_HASH(lport, ipst)];

```

```

1316         if (connp->conn_laddr_v4 != INADDR_ANY) {
1317             IPCL_HASH_INSERT_BOUND(connfp, connp);
1318         } else {
1319             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1320         }
1321         if (cl_inet_listen != NULL) {
1322             ASSERT(connp->conn_ipversion == IPV4_VERSION);
1323             connp->conn_flags |= IPCL_CL_LISTENER;
1324             (*cl_inet_listen)(
1325                 connp->conn_netstack->netstack_stackid,
1326                 IPPROTO_TCP, AF_INET,
1327                 (uint8_t *)&connp->conn_bound_addr_v4, lport, NULL);
1328         }
1329         break;
1330
1331     case IPPROTO_SCTP:
1332         ret = ipcl_sctp_hash_insert(connp, lport);
1333         break;
1334
1335     case IPPROTO_DCCP:
1336         cmn_err(CE_NOTE, "ipclassifier.c: ipcl_bind_insert_v4");
1337         ASSERT(connp->conn_zoneid != ALL_ZONES);
1338         connfp = &ipst->ips_ipcl_dccp_bind_fanout[
1339             IPCL_DCCP_BIND_HASH(lport, ipst)];
1340         if (connp->conn_laddr_v4 != INADDR_ANY) {
1341             IPCL_HASH_INSERT_BOUND(connfp, connp);
1342         } else {
1343             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1344         }
1345         break;
1346 #endif /* ! codereview */
1347 }
1348
1349 #endif /* ! codereview */
1350
1351 return (ret);
1352 }
1353
1354 int
1355 ipcl_bind_insert_v6(conn_t *connp)
1356 {
1357     connf_t      *connfp;
1358     int           ret = 0;
1359     ip_stack_t    *ipst = connp->conn_netstack->netstack_ip;
1360     uint16_t      lport = connp->conn_lport;
1361     uint8_t        protocol = connp->conn_proto;
1362
1363     if (IPCL_IS_IPTUN(connp)) {
1364         return (ipcl_iptun_hash_insert_v6(connp, ipst));
1365     }
1366
1367     switch (protocol) {
1368     default:
1369         if (is_system_labeled() &&
1370             check_exempt_conflict_v6(connp, ipst))
1371             return (EADDRINUSE);
1372         /* FALLTHROUGH */
1373     case IPPROTO_UDP:
1374         if (protocol == IPPROTO_UDP) {
1375             connfp = &ipst->ips_ipcl_udp_fanout[
1376                 IPCL_UDP_HASH(lport, ipst)];
1377         } else {
1378             connfp = &ipst->ips_ipcl_proto_fanout_v6[protocol];
1379         }
1380
1381         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6)) {

```

```

1382             IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1383         } else if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1384             IPCL_HASH_INSERT_BOUND(connfp, connp);
1385         } else {
1386             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1387         }
1388         break;
1389
1390     case IPPROTO_TCP:
1391         /* Insert it in the Bind Hash */
1392         ASSERT(connp->conn_zoneid != ALL_ZONES);
1393         connfp = &ipst->ips_ipcl_bind_fanout[
1394             IPCL_BIND_HASH(lport, ipst)];
1395         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1396             IPCL_HASH_INSERT_BOUND(connfp, connp);
1397         } else {
1398             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1399         }
1400         if (cl_inet_listen != NULL) {
1401             sa_family_t  addr_family;
1402             uint8_t        *laddrp;
1403
1404             if (connp->conn_ipversion == IPV6_VERSION) {
1405                 addr_family = AF_INET6;
1406                 laddrp =
1407                     (uint8_t *)&connp->conn_bound_addr_v6;
1408             } else {
1409                 addr_family = AF_INET;
1410                 laddrp = (uint8_t *)&connp->conn_bound_addr_v4;
1411             }
1412             connp->conn_flags |= IPCL_CL_LISTENER;
1413             (*cl_inet_listen)(
1414                 connp->conn_netstack->netstack_stackid,
1415                 IPPROTO_TCP, addr_family, laddrp, lport, NULL);
1416         }
1417         break;
1418
1419     case IPPROTO_SCTP:
1420         ret = ipcl_sctp_hash_insert(connp, lport);
1421         break;
1422
1423     case IPPROTO_DCCP:
1424         cmn_err(CE_NOTE, "ipclassifier.c: ipcl_bind_insert_v6");
1425         ASSERT(connp->conn_zoneid != ALL_ZONES);
1426         connfp = &ipst->ips_ipcl_dccp_bind_fanout[
1427             IPCL_DCCP_BIND_HASH(lport, ipst)];
1428         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1429             IPCL_HASH_INSERT_BOUND(connfp, connp);
1430         } else {
1431             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1432         }
1433         break;
1434 #endif /* ! codereview */
1435 }
1436
1437 return (ret);
1438 }
1439
1440 /*
1441  * ipcl_conn_hash insertion routines.
1442  * The caller has already set conn_proto and the addresses/ports in the conn_t.
1443  */
1444
1445 int
1446 ipcl_conn_insert(conn_t *connp)
1447 {

```

```

1448     if (connp->conn_ipversion == IPV6_VERSION)
1449         return (ipcl_conn_insert_v6(connp));
1450     else
1451         return (ipcl_conn_insert_v4(connp));
1452 }

1454 int
1455 ipcl_conn_insert_v4(conn_t *connp)
1456 {
1457     connf_t      *connfp;
1458     conn_t        *tconnp;
1459     int           ret = 0;
1460     ip_stack_t    *ipst = connp->conn_netstack->netstack_ip;
1461     uint16_t      lport = connp->conn_lport;
1462     uint8_t       protocol = connp->conn_proto;

1464     if (IPCL_IS_IPTUN(connp))
1465         return (ipcl_iptun_hash_insert(connp, ipst));

1467     switch (protocol) {
1468     case IPPROTO_TCP:
1469         /*
1470          * For TCP, we check whether the connection tuple already
1471          * exists before allowing the connection to proceed. We
1472          * also allow indexing on the zoneid. This is to allow
1473          * multiple shared stack zones to have the same tcp
1474          * connection tuple. In practice this only happens for
1475          * INADDR_LOOPBACK as it's the only local address which
1476          * doesn't have to be unique.
1477          */
1478         connfp = &ipst->ips_ipcl_conn_fanout[
1479             IPCL_CONN_HASH(connp->conn_faddr_v4,
1480                 connp->conn_ports, ipst)];
1481         mutex_enter(&connfp->connf_lock);
1482         for (tconnp = connfp->connf_head; tconnp != NULL;
1483             tconnp = tconnp->conn_next) {
1484             if (IPCL_CONN_MATCH(tconnp, connp->conn_proto,
1485                 connp->conn_faddr_v4, connp->conn_laddr_v4,
1486                 connp->conn_ports) &&
1487                 IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1488                 /* Already have a conn. bail out */
1489                 mutex_exit(&connfp->connf_lock);
1490                 return (EADDRINUSE);
1491             }
1492         }
1493         if (connp->conn_fanout != NULL) {
1494             /*
1495              * Probably a XTI/TLI application trying to do a
1496              * rebind. Let it happen.
1497              */
1498             mutex_exit(&connfp->connf_lock);
1499             IPCL_HASH_REMOVE(connp);
1500             mutex_enter(&connfp->connf_lock);
1501         }

1503         ASSERT(connp->conn_recv != NULL);
1504         ASSERT(connp->conn_recvicmp != NULL);

1506         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1507         mutex_exit(&connfp->connf_lock);
1508         break;

1510     case IPPROTO_SCTP:
1511         /*
1512          * The raw socket may have already been bound, remove it
1513          * from the hash first.

```

```

1514         */
1515         IPCL_HASH_REMOVE(connp);
1516         ret = ipcl_sctp_hash_insert(connp, lport);
1517         break;

1519     case IPPROTO_DCCP:
1520         cmn_err(CE_NOTE, "ipclassifier.c: ipcl_conn_insert_v4");
1521         connfp = &ipst->ips_ipcl_dccp_conn_fanout[IPCL_DCCP_CONN_HASH(
1522             connp->conn_faddr_v4, connp->conn_ports, ipst)];
1523         mutex_enter(&connfp->connf_lock);
1524         for (tconnp = connfp->connf_head; tconnp != NULL;
1525             tconnp = tconnp->conn_next) {
1526             if (IPCL_CONN_MATCH(tconnp, connp->conn_proto,
1527                 connp->conn_faddr_v4, connp->conn_laddr_v4,
1528                 connp->conn_ports) &&
1529                 IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1530                 /* Already have a conn. bail out */
1531                 mutex_exit(&connfp->connf_lock);
1532                 return (EADDRINUSE);
1533             }
1534         }

1536         /* XXX:DCCP XTI/TLI application? */

1538         ASSERT(connp->conn_recv != NULL);
1539         ASSERT(connp->conn_recvicmp != NULL);

1541         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1542         mutex_exit(&connfp->connf_lock);
1543         break;

1545 #endif /* ! codereview */
1546     default:
1547         /*
1548          * Check for conflicts among MAC exempt bindings. For
1549          * transports with port numbers, this is done by the upper
1550          * level per-transport binding logic. For all others, it's
1551          * done here.
1552          */
1553         if (is_system_labeled() &&
1554             check_exempt_conflict_v4(connp, ipst))
1555             return (EADDRINUSE);
1556         /* FALLTHROUGH */

1558     case IPPROTO_UDP:
1559         if (protocol == IPPROTO_UDP) {
1560             connfp = &ipst->ips_ipcl_udp_fanout[
1561                 IPCL_UDP_HASH(lport, ipst)];
1562         } else {
1563             connfp = &ipst->ips_ipcl_proto_fanout_v4[protocol];
1564         }

1566         if (connp->conn_faddr_v4 != INADDR_ANY) {
1567             IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1568         } else if (connp->conn_laddr_v4 != INADDR_ANY) {
1569             IPCL_HASH_INSERT_BOUND(connfp, connp);
1570         } else {
1571             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1572         }
1573         break;
1574     }

1576     return (ret);
1577 }

1579 int

```

```

1580 ipcl_conn_insert_v6(conn_t *connp)
1581 {
1582     connf_t      *connfp;
1583     conn_t        *tconnp;
1584     int           ret = 0;
1585     ip_stack_t    *ipst = connp->conn_netstack->netstack_ip;
1586     uint16_t      lport = connp->conn_lport;
1587     uint8_t        protocol = connp->conn_proto;
1588     uint_t         ifindex = connp->conn_bound_if;
1589
1590     if (IPCL_IS_IPTUN(connp))
1591         return (ipcl_iptun_hash_insert_v6(connp, ipst));
1592
1593     switch (protocol) {
1594     case IPPROTO_TCP:
1595
1596         /*
1597          * For tcp, we check whether the connection tuple already
1598          * exists before allowing the connection to proceed. We
1599          * also allow indexing on the zoneid. This is to allow
1600          * multiple shared stack zones to have the same tcp
1601          * connection tuple. In practice this only happens for
1602          * ipv6_loopback as it's the only local address which
1603          * doesn't have to be unique.
1604          */
1605         connfp = &ipst->ips_ipcl_conn_fanout[
1606             IPCL_CONN_HASH_V6(connp->conn_faddr_v6, connp->conn_ports,
1607             ipst)];
1608         mutex_enter(&connfp->connf_lock);
1609         for (tconnp = connfp->connf_head; tconnp != NULL;
1610             tconnp = tconnp->conn_next) {
1611             /* NOTE: need to match zoneid. Bug in onnv-gate */
1612             if (IPCL_CONN_MATCH_V6(tconnp, connp->conn_proto,
1613                 connp->conn_faddr_v6, connp->conn_laddr_v6,
1614                 connp->conn_ports) &&
1615                 (tconnp->conn_bound_if == 0 ||
1616                 tconnp->conn_bound_if == ifindex) &&
1617                 IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1618                 /* Already have a conn. bail out */
1619                 mutex_exit(&connfp->connf_lock);
1620                 return (EADDRINUSE);
1621             }
1622         }
1623         if (connp->conn_fanout != NULL) {
1624             /*
1625              * Probably a XTI/TLI application trying to do a
1626              * rebind. Let it happen.
1627              */
1628             mutex_exit(&connfp->connf_lock);
1629             IPCL_HASH_REMOVE(connp);
1630             mutex_enter(&connfp->connf_lock);
1631         }
1632         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1633         mutex_exit(&connfp->connf_lock);
1634         break;
1635
1636     case IPPROTO_SCTP:
1637         IPCL_HASH_REMOVE(connp);
1638         ret = ipcl_sctp_hash_insert(connp, lport);
1639         break;
1640
1641     case IPPROTO_DCCP:
1642         cmn_err(CE_NOTE, "ipclassifier.c: ipcl_conn_insert_v6");
1643         connfp = &ipst->ips_ipcl_dccp_conn_fanout[
1644             IPCL_DCCP_CONN_HASH_V6(connp->conn_faddr_v6,
1645             connp->conn_ports, ipst)];

```

```

1646         mutex_enter(&connfp->connf_lock);
1647         for (tconnp = connfp->connf_head; tconnp != NULL;
1648             tconnp = tconnp->conn_next) {
1649             /* NOTE: need to match zoneid. Bug in onnv-gate */
1650             if (IPCL_CONN_MATCH_V6(tconnp, connp->conn_proto,
1651                 connp->conn_faddr_v6, connp->conn_laddr_v6,
1652                 connp->conn_ports) &&
1653                 (tconnp->conn_bound_if == 0 ||
1654                 tconnp->conn_bound_if == ifindex) &&
1655                 IPCL_ZONE_MATCH(tconnp, connp->conn_zoneid)) {
1656                 /* Already have a conn. bail out */
1657                 mutex_exit(&connfp->connf_lock);
1658                 return (EADDRINUSE);
1659             }
1660         }
1661
1662         /* XXX:DCCP XTI/TLI? */
1663         IPCL_HASH_INSERT_CONNECTED_LOCKED(connfp, connp);
1664         mutex_exit(&connfp->connf_lock);
1665         break;
1666     }
1667 #endif /* ! codereview */
1668     default:
1669         if (is_system_labeled() &&
1670             check_exempt_conflict_v6(connp, ipst))
1671             return (EADDRINUSE);
1672         /* FALLTHROUGH */
1673     case IPPROTO_UDP:
1674         if (protocol == IPPROTO_UDP) {
1675             connfp = &ipst->ips_ipcl_udp_fanout[
1676                 IPCL_UDP_HASH(lport, ipst)];
1677         } else {
1678             connfp = &ipst->ips_ipcl_proto_fanout_v6[protocol];
1679         }
1680
1681         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6)) {
1682             IPCL_HASH_INSERT_CONNECTED(connfp, connp);
1683         } else if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6)) {
1684             IPCL_HASH_INSERT_BOUND(connfp, connp);
1685         } else {
1686             IPCL_HASH_INSERT_WILDCARD(connfp, connp);
1687         }
1688         break;
1689     }
1690
1691     return (ret);
1692 }
1693
1694 /*
1695  * v4 packet classifying function. looks up the fanout table to
1696  * find the conn, the packet belongs to. returns the conn with
1697  * the reference held, null otherwise.
1698  *
1699  * If zoneid is ALL_ZONES, then the search rules described in the "Connection
1700  * Lookup" comment block are applied. Labels are also checked as described
1701  * above. If the packet is from the inside (looped back), and is from the same
1702  * zone, then label checks are omitted.
1703  */
1704 conn_t *
1705 ipcl_classify_v4(mblk_t *mp, uint8_t protocol, uint_t hdr_len,
1706     ip_rcv_attr_t *ira, ip_stack_t *ipst)
1707 {
1708     ipha_t *ipha;
1709     connf_t *connfp, *bind_connfp;
1710     uint16_t lport;
1711     uint16_t fport;

```

```

1712     uint32_t ports;
1713     conn_t *connp;
1714     uint16_t *up;
1715     zoneid_t zoneid = ira->ira_zoneid;

1717     ipha = (ipha_t *)mp->b_rptr;
1718     up = (uint16_t *)((uchar_t *)ipha + hdr_len + TCP_PORTS_OFFSET);

1720     switch (protocol) {
1721     case IPPROTO_TCP:
1722         ports = *(uint32_t *)up;
1723         connfp =
1724             &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_src,
1725             ports, ipst)];
1726         mutex_enter(&connfp->connf_lock);
1727         for (connp = connfp->connf_head; connp != NULL;
1728             connp = connp->conn_next) {
1729             if (IPCL_CONN_MATCH(connp, protocol,
1730                 ipha->ipha_src, ipha->ipha_dst, ports) &&
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                 (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1736                 break;
1737         }

1739         if (connp != NULL) {
1740             /*
1741              * We have a fully-bound TCP connection.
1742              *
1743              * For labeled systems, there's no need to check the
1744              * label here. It's known to be good as we checked
1745              * before allowing the connection to become bound.
1746              */
1747             CONN_INC_REF(connp);
1748             mutex_exit(&connfp->connf_lock);
1749             return (connp);
1750         }

1752         mutex_exit(&connfp->connf_lock);
1753         lport = up[1];
1754         bind_connfp =
1755             &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
1756         mutex_enter(&bind_connfp->connf_lock);
1757         for (connp = bind_connfp->connf_head; connp != NULL;
1758             connp = connp->conn_next) {
1759             if (IPCL_BIND_MATCH(connp, protocol, ipha->ipha_dst,
1760                 lport) &&
1761                 (connp->conn_zoneid == zoneid ||
1762                 connp->conn_allzones ||
1763                 ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1764                 (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1765                 (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1766                 break;
1767         }

1769         /*
1770          * If the matching connection is SLP on a private address, then
1771          * the label on the packet must match the local zone's label.
1772          * Otherwise, it must be in the label range defined by tnrx.
1773          * This is ensured by tsol_receive_local.
1774          *
1775          * Note that we don't check tsol_receive_local for
1776          * the connected case.
1777          */

```

```

1778         if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1779             !tsol_receive_local(mp, &ipha->ipha_dst, IPV4_VERSION,
1780             ira, connp)) {
1781             DTRACE_PROBE3(tx_ip_log_info_classify_tcp,
1782                 char *, "connp(1) could not receive mp(2)",
1783                 conn_t *, connp, mblk_t *, mp);
1784             connp = NULL;
1785         }

1787         if (connp != NULL) {
1788             /* Have a listener at least */
1789             CONN_INC_REF(connp);
1790             mutex_exit(&bind_connfp->connf_lock);
1791             return (connp);
1792         }

1794         mutex_exit(&bind_connfp->connf_lock);
1795         break;

1797     case IPPROTO_UDP:
1798         lport = up[1];
1799         fport = up[0];
1800         connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
1801         mutex_enter(&connfp->connf_lock);
1802         for (connp = connfp->connf_head; connp != NULL;
1803             connp = connp->conn_next) {
1804             if (IPCL_UDP_MATCH(connp, lport, ipha->ipha_dst,
1805                 fport, ipha->ipha_src) &&
1806                 (connp->conn_zoneid == zoneid ||
1807                 connp->conn_allzones ||
1808                 ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1809                 (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE))))
1810                 break;
1811         }

1813         if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
1814             !tsol_receive_local(mp, &ipha->ipha_dst, IPV4_VERSION,
1815             ira, connp)) {
1816             DTRACE_PROBE3(tx_ip_log_info_classify_udp,
1817                 char *, "connp(1) could not receive mp(2)",
1818                 conn_t *, connp, mblk_t *, mp);
1819             connp = NULL;
1820         }

1822         if (connp != NULL) {
1823             CONN_INC_REF(connp);
1824             mutex_exit(&connfp->connf_lock);
1825             return (connp);
1826         }

1828         /*
1829          * We shouldn't come here for multicast/broadcast packets
1830          */
1831         mutex_exit(&connfp->connf_lock);

1833         break;

1835     case IPPROTO_DCCP:
1836         ports = *(uint32_t *)up;

1838         /*
1839          * Search for fully-bound connection.
1840          */
1841         connfp = &ipst->ips_ipcl_dccp_conn_fanout[IPCL_DCCP_CONN_HASH(
1842             ipha->ipha_src, ports, ipst)];
1843         mutex_enter(&connfp->connf_lock);

```

```

1844     for (connp = connfp->connf_head; connp != NULL;
1845          connp = connp->conn_next) {
1846         /* XXX:DCCP */
1847         if (IPCL_CONN_MATCH(connp, protocol,
1848             ipha->ipha_src, ipha->ipha_dst, ports)) {
1849             /* XXX */
1850             cmn_err(CE_NOTE, "ipclassifier.c: fully bound co
1851             break;
1852         }
1853     }

1855     if (connp != NULL) {
1856         /*
1857          * We have a fully-bound DCCP connection.
1858          */
1859         CONN_INC_REF(connp);
1860         mutex_exit(&connfp->connf_lock);
1861         return (connp);
1862     }

1864     mutex_exit(&connfp->connf_lock);
1865     lport = up[1];

1867     /*
1868      * Fully-bound connection was not found, search for listener.
1869      */
1870     bind_connfp = &ipst->ips_ipcl_dccp_bind_fanout[
1871         IPCL_DCCP_BIND_HASH(lport, ipst)];
1872     mutex_enter(&bind_connfp->connf_lock);
1873     for (connp = bind_connfp->connf_head; connp != NULL;
1874          connp = connp->conn_next) {
1875         if (IPCL_BIND_MATCH(connp, protocol, ipha->ipha_dst,
1876             lport) &&
1877             (connp->conn_zoneid == zoneid ||
1878             connp->conn_allzones ||
1879             ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1880             (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1881             (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1882             break;
1883     }

1885     if (connp != NULL) {
1886         cmn_err(CE_NOTE, "ipclassifier.c: half-bound bind listen
1887         /* Have a listener at least */
1888         CONN_INC_REF(connp);
1889         mutex_exit(&bind_connfp->connf_lock);
1890         return (connp);
1891     }

1893     mutex_exit(&bind_connfp->connf_lock);
1894     break;

1896 #endif /* ! codereview */
1897     case IPPROTO_ENCAP:
1898     case IPPROTO_IPV6:
1899         return (ipcl iptun_classify_v4(&ipha->ipha_src,
1900             &ipha->ipha_dst, ipst));
1901     }

1903     return (NULL);
1904 }

1906 conn_t *
1907 ipcl_classify_v6(mblk_t *mp, uint8_t protocol, uint_t hdr_len,
1908     ip_rcv_attr_t *ira, ip_stack_t *ipst)
1909 {

```

```

1910     ip6_t      *ip6h;
1911     connf_t     *connfp, *bind_connfp;
1912     uint16_t    lport;
1913     uint16_t    fport;
1914     tcpha_t     *tcpha;
1915     uint32_t    ports;
1916     conn_t      *connp;
1917     uint16_t     *up;
1918     zoneid_t    zoneid = ira->ira_zoneid;

1920     ip6h = (ip6_t *)mp->b_rptr;

1922     switch (protocol) {
1923     case IPPROTO_TCP:
1924         tcpha = (tcpha_t *)&mp->b_rptr[hdr_len];
1925         up = &tcpha->tha_lport;
1926         ports = *(uint32_t *)up;

1928         connfp =
1929             &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_src,
1930                 ports, ipst)];
1931         mutex_enter(&connfp->connf_lock);
1932         for (connp = connfp->connf_head; connp != NULL;
1933              connp = connp->conn_next) {
1934             if (IPCL_CONN_MATCH_V6(connp, protocol,
1935                 ip6h->ip6_src, ip6h->ip6_dst, ports) &&
1936                 (connp->conn_zoneid == zoneid ||
1937                 connp->conn_allzones ||
1938                 ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1939                 (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1940                 (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1941                 break;
1942         }

1944         if (connp != NULL) {
1945             /*
1946              * We have a fully-bound TCP connection.
1947              *
1948              * For labeled systems, there's no need to check the
1949              * label here. It's known to be good as we checked
1950              * before allowing the connection to become bound.
1951              */
1952             CONN_INC_REF(connp);
1953             mutex_exit(&connfp->connf_lock);
1954             return (connp);
1955         }

1957         mutex_exit(&connfp->connf_lock);

1959         lport = up[1];
1960         bind_connfp =
1961             &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
1962         mutex_enter(&bind_connfp->connf_lock);
1963         for (connp = bind_connfp->connf_head; connp != NULL;
1964              connp = connp->conn_next) {
1965             if (IPCL_BIND_MATCH_V6(connp, protocol,
1966                 ip6h->ip6_dst, lport) &&
1967                 (connp->conn_zoneid == zoneid ||
1968                 connp->conn_allzones ||
1969                 ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1970                 (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
1971                 (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
1972                 break;
1973         }

1975         if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&

```



```

1976         !tsol_receive_local(mp, &ip6h->ip6_dst, IPV6_VERSION,
1977         ira, connp)) {
1978             DTRACE_PROBE3(tx_ip_log_info_classify_tcp6,
1979             char *, "connp(1) could not receive mp(2)",
1980             conn_t *, connp, mblk_t *, mp);
1981             connp = NULL;
1982         }
1983
1984         if (connp != NULL) {
1985             /* Have a listener at least */
1986             CONN_INC_REF(connp);
1987             mutex_exit(&bind_connp->connf_lock);
1988             return (connp);
1989         }
1990
1991         mutex_exit(&bind_connp->connf_lock);
1992         break;
1993
1994     case IPPROTO_UDP:
1995         up = (uint16_t *)&mp->b_rptr[hdr_len];
1996         lport = up[1];
1997         fport = up[0];
1998         connfp = &ipst->ips_ipcl_udp_fanout[IPCL_UDP_HASH(lport, ipst)];
1999         mutex_enter(&connfp->connf_lock);
2000         for (connp = connfp->connf_head; connp != NULL;
2001             connp = connp->conn_next) {
2002             if (IPCL_UDP_MATCH_V6(connp, lport, ip6h->ip6_dst,
2003             fport, ip6h->ip6_src) &&
2004             (connp->conn_zoneid == zoneid ||
2005             connp->conn_allzones ||
2006             ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
2007             (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
2008             (ira->ira_flags & IRAF_TX_SHARED_ADDR))))
2009                 break;
2010         }
2011
2012         if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2013         !tsol_receive_local(mp, &ip6h->ip6_dst, IPV6_VERSION,
2014         ira, connp)) {
2015             DTRACE_PROBE3(tx_ip_log_info_classify_udp6,
2016             char *, "connp(1) could not receive mp(2)",
2017             conn_t *, connp, mblk_t *, mp);
2018             connp = NULL;
2019         }
2020
2021         if (connp != NULL) {
2022             CONN_INC_REF(connp);
2023             mutex_exit(&connfp->connf_lock);
2024             return (connp);
2025         }
2026
2027         /*
2028         * We shouldn't come here for multicast/broadcast packets
2029         */
2030         mutex_exit(&connfp->connf_lock);
2031         break;
2032     case IPPROTO_ENCAP:
2033     case IPPROTO_IPV6:
2034         return (ipcl_iptun_classify_v6(&ip6h->ip6_src,
2035         &ip6h->ip6_dst, ipst));
2036     }
2037
2038     return (NULL);
2039 }
2040
2041 /*

```

```

2042 * wrapper around ipcl_classify_(v4,v6) routines.
2043 */
2044 conn_t *
2045 ipcl_classify(mblk_t *mp, ip_recv_attr_t *ira, ip_stack_t *ipst)
2046 {
2047     if (ira->ira_flags & IRAF_IS_IPV4) {
2048         return (ipcl_classify_v4(mp, ira->ira_protocol,
2049         ira->ira_ip_hdr_length, ira, ipst));
2050     } else {
2051         return (ipcl_classify_v6(mp, ira->ira_protocol,
2052         ira->ira_ip_hdr_length, ira, ipst));
2053     }
2054 }
2055
2056 /*
2057 * Only used to classify SCTP RAW sockets
2058 */
2059 conn_t *
2060 ipcl_classify_raw(mblk_t *mp, uint8_t protocol, uint32_t ports,
2061 ipha_t *ipha, ip6_t *ip6h, ip_recv_attr_t *ira, ip_stack_t *ipst)
2062 {
2063     connf_t      *connfp;
2064     conn_t        *connp;
2065     in_port_t     lport;
2066     int           ipversion;
2067     const void    *dst;
2068     zoneid_t      zoneid = ira->ira_zoneid;
2069
2070     lport = ((uint16_t *)&ports)[1];
2071     if (ira->ira_flags & IRAF_IS_IPV4) {
2072         dst = (const void *)&ipha->ipha_dst;
2073         ipversion = IPV4_VERSION;
2074     } else {
2075         dst = (const void *)&ip6h->ip6_dst;
2076         ipversion = IPV6_VERSION;
2077     }
2078
2079     connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(ntohs(lport), ipst)];
2080     mutex_enter(&connfp->connf_lock);
2081     for (connp = connfp->connf_head; connp != NULL;
2082         connp = connp->conn_next) {
2083         /* We don't allow v4 fallback for v6 raw socket. */
2084         if (ipversion != connp->conn_ipversion)
2085             continue;
2086         if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
2087         !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
2088             if (ipversion == IPV4_VERSION) {
2089                 if (!IPCL_CONN_MATCH(connp, protocol,
2090                 ipha->ipha_src, ipha->ipha_dst, ports))
2091                     continue;
2092             } else {
2093                 if (!IPCL_CONN_MATCH_V6(connp, protocol,
2094                 ip6h->ip6_src, ip6h->ip6_dst, ports))
2095                     continue;
2096             }
2097         } else {
2098             if (ipversion == IPV4_VERSION) {
2099                 if (!IPCL_BIND_MATCH(connp, protocol,
2100                 ipha->ipha_dst, lport))
2101                     continue;
2102             } else {
2103                 if (!IPCL_BIND_MATCH_V6(connp, protocol,
2104                 ip6h->ip6_dst, lport))
2105                     continue;
2106             }
2107         }
2108     }

```

```

2109         if (connp->conn_zoneid == zoneid ||
2110             connp->conn_allzones ||
2111             ((connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
2112              (ira->ira_flags & IRAF_TX_MAC_EXEMPTABLE) &&
2113              (ira->ira_flags & IRAF_TX_SHARED_ADDR)))
2114             break;
2115     }

2117     if (connp != NULL && (ira->ira_flags & IRAF_SYSTEM_LABELED) &&
2118         !tsol_receive_local(mp, dst, ipversion, ira, connp)) {
2119         DTRACE_PROBE3(tx_ip_log_info_classify_rawip,
2120             char *, "connp(1) could not receive mp(2)",
2121             conn_t *, connp, mblk_t *, mp);
2122         connp = NULL;
2123     }

2125     if (connp != NULL)
2126         goto found;
2127     mutex_exit(&connfp->connf_lock);

2129     /* Try to look for a wildcard SCTP RAW socket match. */
2130     connfp = &ipst->ips_ipcl_raw_fanout[IPCL_RAW_HASH(0, ipst)];
2131     mutex_enter(&connfp->connf_lock);
2132     for (connp = connfp->connf_head; connp != NULL;
2133         connp = connp->conn_next) {
2134         /* We don't allow v4 fallback for v6 raw socket. */
2135         if (ipversion != connp->conn_ipversion)
2136             continue;
2137         if (!IPCL_ZONE_MATCH(connp, zoneid))
2138             continue;

2140         if (ipversion == IPV4_VERSION) {
2141             if (IPCL_RAW_MATCH(connp, protocol, ipha->ipha_dst))
2142                 break;
2143         } else {
2144             if (IPCL_RAW_MATCH_V6(connp, protocol, ip6h->ip6_dst)) {
2145                 break;
2146             }
2147         }
2148     }

2150     if (connp != NULL)
2151         goto found;

2153     mutex_exit(&connfp->connf_lock);
2154     return (NULL);

2156 found:
2157     ASSERT(connp != NULL);
2158     CONN_INC_REF(connp);
2159     mutex_exit(&connfp->connf_lock);
2160     return (connp);
2161 }

2163 /* ARGSUSED */
2164 static int
2165 tcp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2166 {
2167     itc_t *itc = (itc_t *)buf;
2168     conn_t *connp = &itc->itc_conn;
2169     tcp_t *tcp = (tcp_t *)&itc[1];

2171     bzero(connp, sizeof (conn_t));
2172     bzero(tcp, sizeof (tcp_t));

```

```

2174     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2175     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2176     cv_init(&connp->conn_sq_cv, NULL, CV_DEFAULT, NULL);
2177     tcp->tcp_timer_cache = tcp_timermp_alloc(kmflags);
2178     if (tcp->tcp_timer_cache == NULL)
2179         return (ENOMEM);
2180     connp->conn_tcp = tcp;
2181     connp->conn_flags = IPCL_TCPCONN;
2182     connp->conn_proto = IPPROTO_TCP;
2183     tcp->tcp_connp = connp;
2184     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2186     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2187     if (connp->conn_ixa == NULL) {
2188         tcp_timermp_free(tcp);
2189         return (ENOMEM);
2190     }
2191     connp->conn_ixa->ixa_refcnt = 1;
2192     connp->conn_ixa->ixa_protocol = connp->conn_proto;
2193     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2194     return (0);
2195 }

2197 /* ARGSUSED */
2198 static void
2199 tcp_conn_destructor(void *buf, void *cdrarg)
2200 {
2201     itc_t *itc = (itc_t *)buf;
2202     conn_t *connp = &itc->itc_conn;
2203     tcp_t *tcp = (tcp_t *)&itc[1];

2205     ASSERT(connp->conn_flags & IPCL_TCPCONN);
2206     ASSERT(tcp->tcp_connp == connp);
2207     ASSERT(connp->conn_tcp == tcp);
2208     tcp_timermp_free(tcp);
2209     mutex_destroy(&connp->conn_lock);
2210     cv_destroy(&connp->conn_cv);
2211     cv_destroy(&connp->conn_sq_cv);
2212     rw_destroy(&connp->conn_ilg_lock);

2214     /* Can be NULL if constructor failed */
2215     if (connp->conn_ixa != NULL) {
2216         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2217         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2218         ASSERT(connp->conn_ixa->ixa_nce == NULL);
2219         ixa_refrele(connp->conn_ixa);
2220     }
2221 }

2223 /* ARGSUSED */
2224 static int
2225 ip_conn_constructor(void *buf, void *cdrarg, int kmflags)
2226 {
2227     itc_t *itc = (itc_t *)buf;
2228     conn_t *connp = &itc->itc_conn;

2230     bzero(connp, sizeof (conn_t));
2231     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2232     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2233     connp->conn_flags = IPCL_IPCONN;
2234     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2236     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2237     if (connp->conn_ixa == NULL)
2238         return (ENOMEM);
2239     connp->conn_ixa->ixa_refcnt = 1;

```

```

2240     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2241     return (0);
2242 }

2244 /* ARGSUSED */
2245 static void
2246 ip_conn_destructor(void *buf, void *cdrarg)
2247 {
2248     itc_t    *itc = (itc_t *)buf;
2249     conn_t    *connp = &itc->itc_conn;

2251     ASSERT(connp->conn_flags & IPCL_IPCONN);
2252     ASSERT(connp->conn_priv == NULL);
2253     mutex_destroy(&connp->conn_lock);
2254     cv_destroy(&connp->conn_cv);
2255     rw_destroy(&connp->conn_ilg_lock);

2257     /* Can be NULL if constructor failed */
2258     if (connp->conn_ixa != NULL) {
2259         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2260         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2261         ASSERT(connp->conn_ixa->ixa_nce == NULL);
2262         ixa_refrele(connp->conn_ixa);
2263     }
2264 }

2266 /* ARGSUSED */
2267 static int
2268 udp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2269 {
2270     itc_t    *itc = (itc_t *)buf;
2271     conn_t    *connp = &itc->itc_conn;
2272     udp_t    *udp = (udp_t *)&itc[1];

2274     bzero(connp, sizeof (conn_t));
2275     bzero(udp, sizeof (udp_t));

2277     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2278     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2279     connp->conn_udp = udp;
2280     connp->conn_flags = IPCL_UDPCONN;
2281     connp->conn_proto = IPPROTO_UDP;
2282     udp->udp_connp = connp;
2283     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
2284     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2285     if (connp->conn_ixa == NULL)
2286         return (ENOMEM);
2287     connp->conn_ixa->ixa_refcnt = 1;
2288     connp->conn_ixa->ixa_protocol = connp->conn_proto;
2289     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2290     return (0);
2291 }

2293 /* ARGSUSED */
2294 static void
2295 udp_conn_destructor(void *buf, void *cdrarg)
2296 {
2297     itc_t    *itc = (itc_t *)buf;
2298     conn_t    *connp = &itc->itc_conn;
2299     udp_t    *udp = (udp_t *)&itc[1];

2301     ASSERT(connp->conn_flags & IPCL_UDPCONN);
2302     ASSERT(udp->udp_connp == connp);
2303     ASSERT(connp->conn_udp == udp);
2304     mutex_destroy(&connp->conn_lock);
2305     cv_destroy(&connp->conn_cv);

```

```

2306     rw_destroy(&connp->conn_ilg_lock);

2308     /* Can be NULL if constructor failed */
2309     if (connp->conn_ixa != NULL) {
2310         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2311         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2312         ASSERT(connp->conn_ixa->ixa_nce == NULL);
2313         ixa_refrele(connp->conn_ixa);
2314     }
2315 }

2317 /* ARGSUSED */
2318 static int
2319 rawip_conn_constructor(void *buf, void *cdrarg, int kmflags)
2320 {
2321     itc_t    *itc = (itc_t *)buf;
2322     conn_t    *connp = &itc->itc_conn;
2323     icmp_t    *icmp = (icmp_t *)&itc[1];

2325     bzero(connp, sizeof (conn_t));
2326     bzero(icmp, sizeof (icmp_t));

2328     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2329     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2330     connp->conn_icmp = icmp;
2331     connp->conn_flags = IPCL_RAWIPCONN;
2332     connp->conn_proto = IPPROTO_ICMP;
2333     icmp->icmp_connp = connp;
2334     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
2335     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2336     if (connp->conn_ixa == NULL)
2337         return (ENOMEM);
2338     connp->conn_ixa->ixa_refcnt = 1;
2339     connp->conn_ixa->ixa_protocol = connp->conn_proto;
2340     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2341     return (0);
2342 }

2344 /* ARGSUSED */
2345 static void
2346 rawip_conn_destructor(void *buf, void *cdrarg)
2347 {
2348     itc_t    *itc = (itc_t *)buf;
2349     conn_t    *connp = &itc->itc_conn;
2350     icmp_t    *icmp = (icmp_t *)&itc[1];

2352     ASSERT(connp->conn_flags & IPCL_RAWIPCONN);
2353     ASSERT(icmp->icmp_connp == connp);
2354     ASSERT(connp->conn_icmp == icmp);
2355     mutex_destroy(&connp->conn_lock);
2356     cv_destroy(&connp->conn_cv);
2357     rw_destroy(&connp->conn_ilg_lock);

2359     /* Can be NULL if constructor failed */
2360     if (connp->conn_ixa != NULL) {
2361         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2362         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2363         ASSERT(connp->conn_ixa->ixa_nce == NULL);
2364         ixa_refrele(connp->conn_ixa);
2365     }
2366 }

2368 /* ARGSUSED */
2369 static int
2370 rts_conn_constructor(void *buf, void *cdrarg, int kmflags)
2371 {

```

```

2372     itc_t    *itc = (itc_t *)buf;
2373     conn_t    *connp = &itc->itc_conn;
2374     rts_t     *rts = (rts_t *)&itc[1];

2376     bzero(connp, sizeof (conn_t));
2377     bzero(rts, sizeof (rts_t));

2379     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2380     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2381     connp->conn_rts = rts;
2382     connp->conn_flags = IPCL_RTSCONN;
2383     rts->rts_connp = connp;
2384     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);
2385     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2386     if (connp->conn_ixa == NULL)
2387         return (ENOMEM);
2388     connp->conn_ixa->ixa_refcnt = 1;
2389     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);
2390     return (0);
2391 }

2393 /* ARGSUSED */
2394 static void
2395 rts_conn_destructor(void *buf, void *cdrarg)
2396 {
2397     itc_t    *itc = (itc_t *)buf;
2398     conn_t    *connp = &itc->itc_conn;
2399     rts_t     *rts = (rts_t *)&itc[1];

2401     ASSERT(connp->conn_flags & IPCL_RTSCONN);
2402     ASSERT(rts->rts_connp == connp);
2403     ASSERT(connp->conn_rts == rts);
2404     mutex_destroy(&connp->conn_lock);
2405     cv_destroy(&connp->conn_cv);
2406     rw_destroy(&connp->conn_ilg_lock);

2408     /* Can be NULL if constructor failed */
2409     if (connp->conn_ixa != NULL) {
2410         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2411         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2412         ASSERT(connp->conn_ixa->ixa_nce == NULL);
2413         ixa_refrele(connp->conn_ixa);
2414     }
2415 }

2417 /* ARGSUSED */
2418 static int
2419 dccp_conn_constructor(void *buf, void *cdrarg, int kmflags)
2420 {
2421     itc_t    *itc = (itc_t *)buf;
2422     conn_t    *connp = &itc->itc_conn;
2423     dccp_t    *dccp = (dccp_t *)&itc[1];

2425     bzero(connp, sizeof (conn_t));
2426     bzero(dccp, sizeof (dccp_t));

2428     mutex_init(&connp->conn_lock, NULL, MUTEX_DEFAULT, NULL);
2429     cv_init(&connp->conn_cv, NULL, CV_DEFAULT, NULL);
2430     rw_init(&connp->conn_ilg_lock, NULL, RW_DEFAULT, NULL);

2432     dccp->dccp_timercache = dccp_timermp_alloc(kmflags);
2433     if (dccp->dccp_timercache == NULL) {
2434         return (ENOMEM);
2435     }

2437     connp->conn_dccp = dccp;

```

```

2438     connp->conn_flags = IPCL_DCCPCONN;
2439     connp->conn_proto = IPPROTO_DCCP;
2440     dccp->dccp_connp = connp;

2442     connp->conn_ixa = kmem_zalloc(sizeof (ip_xmit_attr_t), kmflags);
2443     if (connp->conn_ixa == NULL) {
2444         return (NULL);
2445     }

2447     connp->conn_ixa->ixa_refcnt = 1;
2448     connp->conn_ixa->ixa_protocol = connp->conn_proto;
2449     connp->conn_ixa->ixa_xmit_hint = CONN_TO_XMIT_HINT(connp);

2451     return (0);
2452 }

2454 /* ARGSUSED */
2455 static void
2456 dccp_conn_destructor(void *buf, void *cdrarg)
2457 {
2458     itc_t    *itc = (itc_t *)buf;
2459     conn_t    *connp = &itc->itc_conn;
2460     dccp_t    *dccp = (dccp_t *)&itc[1];

2462     ASSERT(connp->conn_flags & IPCL_DCCPCONN);
2463     ASSERT(dccp->dccp_connp == connp);
2464     ASSERT(connp->conn_dccp == dccp);

2466     dccp_timermp_free(dccp);

2468     mutex_destroy(&connp->conn_lock);
2469     cv_destroy(&connp->conn_cv);
2470     rw_destroy(&connp->conn_ilg_lock);

2472     if (connp->conn_ixa != NULL) {
2473         ASSERT(connp->conn_ixa->ixa_refcnt == 1);
2474         ASSERT(connp->conn_ixa->ixa_ire == NULL);
2475         ASSERT(connp->conn_ixa->ixa_nce == NULL);

2477         ixa_refrele(connp->conn_ixa);
2478     }
2479 }

2481 #endif /* ! codereview */
2482 /*
2483  * Called as part of ipcl_conn_destroy to assert and clear any pointers
2484  * in the conn_t.
2485  *
2486  * Below we list all the pointers in the conn_t as a documentation aid.
2487  * The ones that we can not ASSERT to be NULL are #ifdef'ed out.
2488  * If you add any pointers to the conn_t please add an ASSERT here
2489  * and #ifdef it out if it can't be actually asserted to be NULL.
2490  * In any case, we bzero most of the conn_t at the end of the function.
2491  */
2492 void
2493 ipcl_conn_cleanup(conn_t *connp)
2494 {
2495     ip_xmit_attr_t *ixa;

2497     ASSERT(connp->conn_latch == NULL);
2498     ASSERT(connp->conn_latch_in_policy == NULL);
2499     ASSERT(connp->conn_latch_in_action == NULL);
2500 #ifndef notdef
2501     ASSERT(connp->conn_rq == NULL);
2502     ASSERT(connp->conn_wq == NULL);
2503 #endif

```

```

2504     ASSERT(connp->conn_cred == NULL);
2505     ASSERT(connp->conn_g_fanout == NULL);
2506     ASSERT(connp->conn_g_next == NULL);
2507     ASSERT(connp->conn_g_prev == NULL);
2508     ASSERT(connp->conn_policy == NULL);
2509     ASSERT(connp->conn_fanout == NULL);
2510     ASSERT(connp->conn_next == NULL);
2511     ASSERT(connp->conn_prev == NULL);
2512     ASSERT(connp->conn_oper_pending_ill == NULL);
2513     ASSERT(connp->conn_ilg == NULL);
2514     ASSERT(connp->conn_drain_next == NULL);
2515     ASSERT(connp->conn_drain_prev == NULL);
2516 #ifndef notdef
2517     /* conn_idl is not cleared when removed from idl list */
2518     ASSERT(connp->conn_idl == NULL);
2519 #endif
2520     ASSERT(connp->conn_ipsec_opt_mp == NULL);
2521 #ifndef notdef
2522     /* conn_netstack is cleared by the caller; needed by ixa_cleanup */
2523     ASSERT(connp->conn_netstack == NULL);
2524 #endif

2526     ASSERT(connp->conn_helper_info == NULL);
2527     ASSERT(connp->conn_ixa != NULL);
2528     ixa = connp->conn_ixa;
2529     ASSERT(ixa->ixa_refcnt == 1);
2530     /* Need to preserve ixa_protocol */
2531     ixa_cleanup(ixa);
2532     ixa->ixa_flags = 0;

2534     /* Clear out the conn_t fields that are not preserved */
2535     bzero(&connp->conn_start_clr,
2536         sizeof (conn_t) -
2537         ((uchar_t *)&connp->conn_start_clr - (uchar_t *)connp));
2538 }

2540 /*
2541  * All conns are inserted in a global multi-list for the benefit of
2542  * walkers. The walk is guaranteed to walk all open conns at the time
2543  * of the start of the walk exactly once. This property is needed to
2544  * achieve some cleanups during unplumb of interfaces. This is achieved
2545  * as follows.
2546  *
2547  * ipcl_conn_create and ipcl_conn_destroy are the only functions that
2548  * call the insert and delete functions below at creation and deletion
2549  * time respectively. The conn never moves or changes its position in this
2550  * multi-list during its lifetime. CONN_CONDEMNED ensures that the refcnt
2551  * won't increase due to walkers, once the conn deletion has started. Note
2552  * that we can't remove the conn from the global list and then wait for
2553  * the refcnt to drop to zero, since walkers would then see a truncated
2554  * list. CONN_INCIPIENT ensures that walkers don't start looking at
2555  * conns until ip_open is ready to make them globally visible.
2556  * The global round robin multi-list locks are held only to get the
2557  * next member/insertion/deletion and contention should be negligible
2558  * if the multi-list is much greater than the number of cpus.
2559  */
2560 void
2561 ipcl_globalhash_insert(conn_t *connp)
2562 {
2563     int         index;
2564     struct connf_s *connfp;
2565     ip_stack_t  *ipst = connp->conn_netstack->netstack_ip;

2567     /*
2568      * No need for atomic here. Approximate even distribution
2569      * in the global lists is sufficient.

```

```

2570     /*
2571     ipst->ips_conn_g_index++;
2572     index = ipst->ips_conn_g_index & (CONN_G_HASH_SIZE - 1);

2574     connp->conn_g_prev = NULL;
2575     /*
2576     * Mark as INCIPIENT, so that walkers will ignore this
2577     * for now, till ip_open is ready to make it visible globally.
2578     */
2579     connp->conn_state_flags |= CONN_INCIPIENT;

2581     connfp = &ipst->ips_ipcl_globalhash_fanout[index];
2582     /* Insert at the head of the list */
2583     mutex_enter(&connfp->connf_lock);
2584     connp->conn_g_next = connfp->connf_head;
2585     if (connp->conn_g_next != NULL)
2586         connp->conn_g_next->conn_g_prev = connp;
2587     connfp->connf_head = connp;

2589     /* The fanout bucket this conn points to */
2590     connp->conn_g_fanout = connfp;

2592     mutex_exit(&connfp->connf_lock);
2593 }

2595 void
2596 ipcl_globalhash_remove(conn_t *connp)
2597 {
2598     struct connf_s *connfp;

2600     /*
2601     * We were never inserted in the global multi list.
2602     * IPCL_NONE variety is never inserted in the global multilist
2603     * since it is presumed to not need any cleanup and is transient.
2604     */
2605     if (connp->conn_g_fanout == NULL)
2606         return;

2608     connfp = connp->conn_g_fanout;
2609     mutex_enter(&connfp->connf_lock);
2610     if (connp->conn_g_prev != NULL)
2611         connp->conn_g_prev->conn_g_next = connp->conn_g_next;
2612     else
2613         connfp->connf_head = connp->conn_g_next;
2614     if (connp->conn_g_next != NULL)
2615         connp->conn_g_next->conn_g_prev = connp->conn_g_prev;
2616     mutex_exit(&connfp->connf_lock);

2618     /* Better to stumble on a null pointer than to corrupt memory */
2619     connp->conn_g_next = NULL;
2620     connp->conn_g_prev = NULL;
2621     connp->conn_g_fanout = NULL;
2622 }

2624 /*
2625  * Walk the list of all conn_t's in the system, calling the function provided
2626  * With the specified argument for each.
2627  * Applies to both IPv4 and IPv6.
2628  *
2629  * CONNs may hold pointers to ill's (conn_dhcpinit_ill and
2630  * conn_oper_pending_ill). To guard against stale pointers
2631  * ipcl_walk() is called to cleanup the conn_t's, typically when an interface is
2632  * unplumbed or removed. New conn_t's that are created while we are walking
2633  * may be missed by this walk, because they are not necessarily inserted
2634  * at the tail of the list. They are new conn_t's and thus don't have any
2635  * stale pointers. The CONN_CLOSING flag ensures that no new reference

```

```

2636  * is created to the struct that is going away.
2637  */
2638  void
2639  ipcl_walk(pfv_t func, void *arg, ip_stack_t *ipst)
2640  {
2641      int i;
2642      conn_t *connp;
2643      conn_t *prev_connp;
2644
2645      for (i = 0; i < CONN_G_HASH_SIZE; i++) {
2646          mutex_enter(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2647          prev_connp = NULL;
2648          connp = ipst->ips_ipcl_globalhash_fanout[i].connf_head;
2649          while (connp != NULL) {
2650              mutex_enter(&connp->conn_lock);
2651              if (connp->conn_state_flags &
2652                  (CONN_CONDEMNED | CONN_INCIPIENT)) {
2653                  mutex_exit(&connp->conn_lock);
2654                  connp = connp->conn_g_next;
2655                  continue;
2656              }
2657              CONN_INC_REF_LOCKED(connp);
2658              mutex_exit(&connp->conn_lock);
2659              mutex_exit(
2660                  &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2661              (*func)(connp, arg);
2662              if (prev_connp != NULL)
2663                  CONN_DEC_REF(prev_connp);
2664              mutex_enter(
2665                  &ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2666              prev_connp = connp;
2667              connp = connp->conn_g_next;
2668          }
2669          mutex_exit(&ipst->ips_ipcl_globalhash_fanout[i].connf_lock);
2670          if (prev_connp != NULL)
2671              CONN_DEC_REF(prev_connp);
2672      }
2673  }
2674
2675  /*
2676  * Search for a peer TCP/IPv4 loopback conn by doing a reverse lookup on
2677  * the {src, dst, lport, fport} quadruplet. Returns with conn reference
2678  * held; caller must call CONN_DEC_REF. Only checks for connected entries
2679  * (peer tcp in ESTABLISHED state).
2680  */
2681  conn_t *
2682  ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *connp, ipha_t *ipha, tcpha_t *tcpha,
2683      ip_stack_t *ipst)
2684  {
2685      uint32_t ports;
2686      uint16_t *pports = (uint16_t *)&ports;
2687      connf_t *connfp;
2688      conn_t *tconnp;
2689      boolean_t zone_chk;
2690
2691      /*
2692      * If either the source of destination address is loopback, then
2693      * both endpoints must be in the same Zone. Otherwise, both of
2694      * the addresses are system-wide unique (tcp is in ESTABLISHED
2695      * state) and the endpoints may reside in different Zones.
2696      */
2697      zone_chk = (ipha->ipha_src == htonl(INADDR_LOOPBACK)) ||
2698          ipha->ipha_dst == htonl(INADDR_LOOPBACK));
2699
2700      pports[0] = tcpha->tha_fport;
2701      pports[1] = tcpha->tha_lport;

```

```

2703      connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_dst,
2704          ports, ipst)];
2705
2706      mutex_enter(&connfp->connf_lock);
2707      for (tconnp = connfp->connf_head; tconnp != NULL;
2708          tconnp = tconnp->conn_next) {
2709
2710          if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2711              ipha->ipha_dst, ipha->ipha_src, ports) &&
2712              tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2713              (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {
2714
2715              ASSERT(tconnp != connp);
2716              CONN_INC_REF(tconnp);
2717              mutex_exit(&connfp->connf_lock);
2718              return (tconnp);
2719          }
2720      }
2721      mutex_exit(&connfp->connf_lock);
2722      return (NULL);
2723  }
2724
2725  /*
2726  * Search for a peer TCP/IPv6 loopback conn by doing a reverse lookup on
2727  * the {src, dst, lport, fport} quadruplet. Returns with conn reference
2728  * held; caller must call CONN_DEC_REF. Only checks for connected entries
2729  * (peer tcp in ESTABLISHED state).
2730  */
2731  conn_t *
2732  ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *connp, ip6_t *ip6h, tcpha_t *tcpha,
2733      ip_stack_t *ipst)
2734  {
2735      uint32_t ports;
2736      uint16_t *pports = (uint16_t *)&ports;
2737      connf_t *connfp;
2738      conn_t *tconnp;
2739      boolean_t zone_chk;
2740
2741      /*
2742      * If either the source of destination address is loopback, then
2743      * both endpoints must be in the same Zone. Otherwise, both of
2744      * the addresses are system-wide unique (tcp is in ESTABLISHED
2745      * state) and the endpoints may reside in different Zones. We
2746      * don't do Zone check for link local address(es) because the
2747      * current Zone implementation treats each link local address as
2748      * being unique per system node, i.e. they belong to global Zone.
2749      */
2750      zone_chk = (IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_src) ||
2751          IN6_IS_ADDR_LOOPBACK(&ip6h->ip6_dst));
2752
2753      pports[0] = tcpha->tha_fport;
2754      pports[1] = tcpha->tha_lport;
2755
2756      connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2757          ports, ipst)];
2758
2759      mutex_enter(&connfp->connf_lock);
2760      for (tconnp = connfp->connf_head; tconnp != NULL;
2761          tconnp = tconnp->conn_next) {
2762
2763          /* We skip conn_bound_if check here as this is loopback tcp */
2764          if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2765              ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2766              tconnp->conn_tcp->tcp_state == TCPS_ESTABLISHED &&
2767              (!zone_chk || tconnp->conn_zoneid == connp->conn_zoneid)) {

```

```

2769         ASSERT(tconnp != connp);
2770         CONN_INC_REF(tconnp);
2771         mutex_exit(&connfp->connf_lock);
2772         return (tconnp);
2773     }
2774 }
2775 mutex_exit(&connfp->connf_lock);
2776 return (NULL);
2777 }

2779 /*
2780  * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2781  * Returns with conn reference held. Caller must call CONN_DEC_REF.
2782  * Only checks for connected entries i.e. no INADDR_ANY checks.
2783  */
2784 conn_t *
2785 ipcl_tcp_lookup_reversed_ipv4(ipha_t *ipha, tcpha_t *tcpha, int min_state,
2786 ip_stack_t *ipst)
2787 {
2788     uint32_t ports;
2789     uint16_t *pports;
2790     connf_t *connfp;
2791     conn_t *tconnp;

2793     pports = (uint16_t *)&ports;
2794     pports[0] = tcpha->tha_fport;
2795     pports[1] = tcpha->tha_lport;

2797     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH(ipha->ipha_dst,
2798 ports, ipst)];

2800     mutex_enter(&connfp->connf_lock);
2801     for (tconnp = connfp->connf_head; tconnp != NULL;
2802          tconnp = tconnp->conn_next) {
2804         if (IPCL_CONN_MATCH(tconnp, IPPROTO_TCP,
2805 ipha->ipha_dst, ipha->ipha_src, ports) &&
2806 tconnp->conn_tcp->tcp_state >= min_state) {
2808             CONN_INC_REF(tconnp);
2809             mutex_exit(&connfp->connf_lock);
2810             return (tconnp);
2811         }
2812     }
2813     mutex_exit(&connfp->connf_lock);
2814     return (NULL);
2815 }

2817 /*
2818  * Find an exact {src, dst, lport, fport} match for a bounced datagram.
2819  * Returns with conn reference held. Caller must call CONN_DEC_REF.
2820  * Only checks for connected entries i.e. no INADDR_ANY checks.
2821  * Match on ifindex in addition to addresses.
2822  */
2823 conn_t *
2824 ipcl_tcp_lookup_reversed_ipv6(ip6_t *ip6h, tcpha_t *tcpha, int min_state,
2825 uint_t ifindex, ip_stack_t *ipst)
2826 {
2827     tcp_t *tcp;
2828     uint32_t ports;
2829     uint16_t *pports;
2830     connf_t *connfp;
2831     conn_t *tconnp;

2833     pports = (uint16_t *)&ports;

```

```

2834     pports[0] = tcpha->tha_fport;
2835     pports[1] = tcpha->tha_lport;

2837     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2838 ports, ipst)];

2840     mutex_enter(&connfp->connf_lock);
2841     for (tconnp = connfp->connf_head; tconnp != NULL;
2842          tconnp = tconnp->conn_next) {
2844         tcp = tconnp->conn_tcp;
2845         if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2846 ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2847 tcp->tcp_state >= min_state &&
2848 (tconnp->conn_bound_if == 0 ||
2849 tconnp->conn_bound_if == ifindex)) {
2851             CONN_INC_REF(tconnp);
2852             mutex_exit(&connfp->connf_lock);
2853             return (tconnp);
2854         }
2855     }
2856     mutex_exit(&connfp->connf_lock);
2857     return (NULL);
2858 }

2860 /*
2861  * Same as ipcl_tcp_lookup_reversed_ipv4.
2862  */
2863 conn_t *
2864 ipcl_dccp_lookup_reversed_ipv4(ipha_t *ipha, dccpha_t *dccpha, int min_state,
2865 ip_stack_t *ipst)
2866 {
2867     conn_t *tconnp;
2868     connf_t *connfp;
2869     uint16_t *pports;
2870     uint32_t ports;

2872     pports = (uint16_t *)&ports;
2873     pports[0] = dccpha->dha_fport;
2874     pports[1] = dccpha->dha_lport;

2876     connfp = &ipst->ips_ipcl_dccp_conn_fanout[IPCL_DCCP_CONN_HASH(
2877 ipha->ipha_dst, ports, ipst)];

2879     mutex_enter(&connfp->connf_lock);
2880     for (tconnp = connfp->connf_head; tconnp != NULL;
2881          tconnp = tconnp->conn_next) {
2882         if (IPCL_CONN_MATCH(tconnp, IPPROTO_DCCP,
2883 ipha->ipha_dst, ipha->ipha_src, ports) &&
2884 tconnp->conn_dccp->dccp_state >= min_state) {
2885             CONN_INC_REF(tconnp);
2886             mutex_exit(&connfp->connf_lock);
2887             return (tconnp);
2888         }
2889     }
2890     mutex_exit(&connfp->connf_lock);

2892     return (NULL);
2893 }

2895 /*
2896  * Same as ipcl_tcp_lookup_reversed_ipv6.
2897  */
2898 conn_t *
2899 ipcl_dccp_lookup_reversed_ipv6(ip6_t *ip6h, dccpha_t *dccpha, int min_state,

```

```

2900     uint_t ifindex, ip_stack_t *ipst)
2901 {
2902     conn_t      *tconnp;
2903     tcp_t       *tcp;
2904     connf_t     *connfp;
2905     uint32_t    ports;
2906     uint16_t    *pports;
2907
2908     pports = (uint16_t *)&ports;
2909     pports[0] = dccpha->dha_fport;
2910     pports[1] = dccpha->dha_lport;
2911 /*
2912     connfp = &ipst->ips_ipcl_conn_fanout[IPCL_CONN_HASH_V6(ip6h->ip6_dst,
2913     ports, ipst)];
2914
2915     mutex_enter(&connfp->connf_lock);
2916     for (tconnp = connfp->connf_head; tconnp != NULL;
2917         tconnp = tconnp->conn_next) {
2918
2919         tcp = tconnp->conn_tcp;
2920         if (IPCL_CONN_MATCH_V6(tconnp, IPPROTO_TCP,
2921             ip6h->ip6_dst, ip6h->ip6_src, ports) &&
2922             tcp->tcp_state >= min_state &&
2923             (tconnp->conn_bound_if == 0 ||
2924             tconnp->conn_bound_if == ifindex)) {
2925
2926             CONN_INC_REF(tconnp);
2927             mutex_exit(&connfp->connf_lock);
2928             return (tconnp);
2929         }
2930     }
2931     mutex_exit(&connfp->connf_lock);
2932 */
2933     return (NULL);
2934 }
2935
2936 /*
2937 #endif /* ! codereview */
2938 * Finds a TCP/IPv4 listening connection; called by tcp_disconnect to locate
2939 * a listener when changing state.
2940 */
2941 conn_t *
2942 ipcl_lookup_listener_v4(uint16_t lport, ipaddr_t laddr, zoneid_t zoneid,
2943     ip_stack_t *ipst)
2944 {
2945     connf_t      *bind_connpf;
2946     conn_t       *connp;
2947     tcp_t        *tcp;
2948
2949     /*
2950     * Avoid false matches for packets sent to an IP destination of
2951     * all zeros.
2952     */
2953     if (laddr == 0)
2954         return (NULL);
2955
2956     ASSERT(zoneid != ALL_ZONES);
2957
2958     bind_connpf = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2959     mutex_enter(&bind_connpf->connf_lock);
2960     for (connp = bind_connpf->connf_head; connp != NULL;
2961         connp = connp->conn_next) {
2962         tcp = connp->conn_tcp;
2963         if (IPCL_BIND_MATCH(connp, IPPROTO_TCP, laddr, lport) &&
2964             IPCL_ZONE_MATCH(connp, zoneid) &&
2965             (tcp->tcp_listener == NULL)) {

```

```

2966             CONN_INC_REF(connp);
2967             mutex_exit(&bind_connpf->connf_lock);
2968             return (connp);
2969         }
2970     }
2971     mutex_exit(&bind_connpf->connf_lock);
2972     return (NULL);
2973 }
2974
2975 /*
2976 * Finds a TCP/IPv6 listening connection; called by tcp_disconnect to locate
2977 * a listener when changing state.
2978 */
2979 conn_t *
2980 ipcl_lookup_listener_v6(uint16_t lport, in6_addr_t *laddr, uint_t ifindex,
2981     zoneid_t zoneid, ip_stack_t *ipst)
2982 {
2983     connf_t      *bind_connpf;
2984     conn_t       *connp = NULL;
2985     tcp_t        *tcp;
2986
2987     /*
2988     * Avoid false matches for packets sent to an IP destination of
2989     * all zeros.
2990     */
2991     if (IN6_IS_ADDR_UNSPECIFIED(laddr))
2992         return (NULL);
2993
2994     ASSERT(zoneid != ALL_ZONES);
2995
2996     bind_connpf = &ipst->ips_ipcl_bind_fanout[IPCL_BIND_HASH(lport, ipst)];
2997     mutex_enter(&bind_connpf->connf_lock);
2998     for (connp = bind_connpf->connf_head; connp != NULL;
2999         connp = connp->conn_next) {
3000         tcp = connp->conn_tcp;
3001         if (IPCL_BIND_MATCH_V6(connp, IPPROTO_TCP, *laddr, lport) &&
3002             IPCL_ZONE_MATCH(connp, zoneid) &&
3003             (connp->conn_bound_if == 0 ||
3004             connp->conn_bound_if == ifindex) &&
3005             tcp->tcp_listener == NULL) {
3006             CONN_INC_REF(connp);
3007             mutex_exit(&bind_connpf->connf_lock);
3008             return (connp);
3009         }
3010     }
3011     mutex_exit(&bind_connpf->connf_lock);
3012     return (NULL);
3013 }
3014
3015 /*
3016 * ipcl_get_next_conn
3017 * get the next entry in the conn global list
3018 * and put a reference on the next_conn.
3019 * decrement the reference on the current conn.
3020 *
3021 * This is an iterator based walker function that also provides for
3022 * some selection by the caller. It walks through the conn_hash bucket
3023 * searching for the next valid connp in the list, and selects connections
3024 * that are neither closed nor condemned. It also REFHOLDS the conn
3025 * thus ensuring that the conn exists when the caller uses the conn.
3026 */
3027 conn_t *
3028 ipcl_get_next_conn(connf_t *connfp, conn_t *connp, uint32_t conn_flags)
3029 {
3030     conn_t      *next_connp;

```



```

3032     if (connfp == NULL)
3033         return (NULL);

3035     mutex_enter(&connfp->connf_lock);

3037     next_connp = (connp == NULL) ?
3038         connfp->connf_head : connp->conn_g_next;

3040     while (next_connp != NULL) {
3041         mutex_enter(&next_connp->conn_lock);
3042         if (!(next_connp->conn_flags & conn_flags) ||
3043             (next_connp->conn_state_flags &
3044              (CONN_CONDEMNED | CONN_INCIPIENT))) {
3045             /*
3046              * This conn has been condemned or
3047              * is closing, or the flags don't match
3048              */
3049             mutex_exit(&next_connp->conn_lock);
3050             next_connp = next_connp->conn_g_next;
3051             continue;
3052         }
3053         CONN_INC_REF_LOCKED(next_connp);
3054         mutex_exit(&next_connp->conn_lock);
3055         break;
3056     }

3058     mutex_exit(&connfp->connf_lock);

3060     if (connp != NULL)
3061         CONN_DEC_REF(connp);

3063     return (next_connp);
3064 }

3066 #ifdef CONN_DEBUG
3067 /*
3068  * Trace of the last NBUF refhold/refrele
3069  */
3070 int
3071 conn_trace_ref(conn_t *connp)
3072 {
3073     int     last;
3074     conn_trace_t  *ctb;

3076     ASSERT(MUTEX_HELD(&connp->conn_lock));
3077     last = connp->conn_trace_last;
3078     last++;
3079     if (last == CONN_TRACE_MAX)
3080         last = 0;

3082     ctb = &connp->conn_trace_buf[last];
3083     ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
3084     connp->conn_trace_last = last;
3085     return (1);
3086 }

3088 int
3089 conn_untrace_ref(conn_t *connp)
3090 {
3091     int     last;
3092     conn_trace_t  *ctb;

3094     ASSERT(MUTEX_HELD(&connp->conn_lock));
3095     last = connp->conn_trace_last;
3096     last++;
3097     if (last == CONN_TRACE_MAX)

```

```

3098         last = 0;

3100     ctb = &connp->conn_trace_buf[last];
3101     ctb->ctb_depth = getpcstack(ctb->ctb_stack, CONN_STACK_DEPTH);
3102     connp->conn_trace_last = last;
3103     return (1);
3104 }
3105 #endif

```

new/usr/src/uts/common/inet/ip_impl.h

1

```
*****
6502 Sat Aug 18 10:37:17 2012
new/usr/src/uts/common/inet/ip_impl.h
dccp: reset packet
*****

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(iph, hlen) \
80 ((uint16_t *)(((uchar_t *)iph) + ((hlen) + TCP_CHECKSUM_OFFSET)))

82 #define IPH_UDPH_CHECKSUMP(iph, hlen) \
83 ((uint16_t *)(((uchar_t *)iph) + ((hlen) + UDP_CHECKSUM_OFFSET)))

85 #define IPH_ICMPV6_CHECKSUMP(iph, hlen) \
86 ((uint16_t *)(((uchar_t *)iph) + ((hlen) + ICMPV6_CHECKSUM_OFFSET)))

88 #define IPH_DCCPH_CHECKSUMP(iph, hlen) \
89 ((uint16_t *)(((uchar_t *)iph) + ((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_rcv_attr_t *,
192     zoneid_t);
193 extern zoneid_t ip_get_zoneid_v6(in6_addr_t *, mblk_t *, const ill_t *,

```

```

194     ip_rcv_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 */

```

new/usr/src/uts/common/inet/ip_stack.h

1

```
*****
13785 Sat Aug 18 10:37:17 2012
new/usr/src/uts/common/inet/ip_stack.h
dccp: ips_ipcl_dccp_fanout
*****
_____unchanged_portion_omitted_____

143 /*
144  * IP stack instances
145  */
146 struct ip_stack {
147     netstack_t      *ips_netstack; /* Common netstack */

149     uint_t           ips_src_generation; /* Both IPv4 and IPv6 */

151     struct mod_prop_info_s *ips_propinfo_tbl; /* ip tunables table */

153     mib2_ipIfStatsEntry_t ips_ip_mib; /* SNMP fixed size info */
154     mib2_icmp_t ips_icmp_mib;
155     /*
156      * IPv6 mibs when the interface (ill) is not known.
157      * When the ill is known the per-interface mib in the ill is used.
158      */
159     mib2_ipIfStatsEntry_t ips_ip6_mib;
160     mib2_ipv6IfIcmpEntry_t ips_icmp6_mib;

162     struct igmpstat ips_igmpstat;

164     kstat_t *ips_ip_mibkp; /* kstat exporting ip_mib data */
165     kstat_t *ips_icmp_mibkp; /* kstat exporting icmp_mib data */
166     kstat_t *ips_ip_kstat;
167     ip_stat_t ips_ip_statistics;
168     kstat_t *ips_ip6_kstat;
169     ip6_stat_t ips_ip6_statistics;

171 /* ip.c */
172     kmutex_t ips_igmp_timer_lock;
173     kmutex_t ips_mld_timer_lock;
174     kmutex_t ips_ip_mi_lock;
175     kmutex_t ips_ip_addr_avail_lock;
176     krwlock_t ips_ill_g_lock;

178     krwlock_t ips_ill_g_usersrc_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_conn_fanout;
199     struct connf_s *ips_ipcl_dccp_bind_fanout;
200 #endif /* ! codereview */
201     uint_t ips_ipcl_conn_fanout_size;
```

new/usr/src/uts/common/inet/ip_stack.h

2

```
202     uint_t ips_ipcl_bind_fanout_size;
203     uint_t ips_ipcl_udp_fanout_size;
204     uint_t ips_ipcl_raw_fanout_size;
205     uint_t ips_ipcl iptun_fanout_size;
206     uint_t ips_ipcl_dccp_conn_fanout_size;
207     uint_t ips_ipcl_dccp_bind_fanout_size;
208 #endif /* ! codereview */
209     struct connf_s *ips_ipcl_globalhash_fanout;
210     int ips_conn_g_index;

212 /* ip.c */
213 /* Following protected by igmp_timer_lock */
214     int ips_igmp_time_to_next; /* Time since last timeout */
215     int ips_igmp_timer_scheduled_last;
216     int ips_igmp_deferred_next;
217     timeout_id_t ips_igmp_timeout_id;
218     boolean_t ips_igmp_timer_setter_active;

220 /* Following protected by mld_timer_lock */
221     int ips_mld_time_to_next; /* Time since last timeout */
222     int ips_mld_timer_scheduled_last;
223     int ips_mld_deferred_next;
224     timeout_id_t ips_mld_timeout_id;
225     boolean_t ips_mld_timer_setter_active;

227 /* Protected by igmp_slowtimeout_lock */
228     timeout_id_t ips_igmp_slowtimeout_id;
229     kmutex_t ips_igmp_slowtimeout_lock;

231 /* Protected by mld_slowtimeout_lock */
232     timeout_id_t ips_mld_slowtimeout_id;
233     kmutex_t ips_mld_slowtimeout_lock;

235 /* IPv4 forwarding table */
236     struct radix_node_head *ips_ip_ftable;

238 #define IPV6_ABITS 128
239 #define IPV6_MASK_TABLE_SIZE (IPV6_ABITS + 1) /* 129 ptrs */
240     struct irb *ips_ip_forwarding_table_v6[IPV6_MASK_TABLE_SIZE];

242 /*
243  * ire_ft_init_lock is used while initializing ip_forwarding_table
244  * dynamically in ire_add.
245  */
246     kmutex_t ips_ire_ft_init_lock;

248 /*
249  * This is the IPv6 counterpart of RADIX_NODE_HEAD_LOCK. It is used
250  * to prevent adds and deletes while we are doing a ftable_lookup
251  * and extracting the ire_generation.
252  */
253     krwlock_t ips_ip6_ire_head_lock;

255     uint32_t ips_ip6_ftable_hash_size;

257     ire_stats_t ips_ire_stats_v4; /* IPv4 ire statistics */
258     ire_stats_t ips_ire_stats_v6; /* IPv6 ire statistics */

260 /* Count how many condemned objects for kmem_cache callbacks */
261     uint32_t ips_num_ire_condemned;
262     uint32_t ips_num_nce_condemned;
263     uint32_t ips_num_dce_condemned;

265     struct ire_s *ips_ire_reject_v4; /* For unreachable dests */
266     struct ire_s *ips_ire_reject_v6; /* For unreachable dests */
267     struct ire_s *ips_ire_blackhole_v4; /* For temporary failures */
```

```

268     struct ire_s      *ips_ire_blackhole_v6; /* For temporary failures */
270     /* ips_ire_dep_lock protects ire_dep_* relationship between IRES */
271     krmlock_t          ips_ire_dep_lock;

273     /* Destination Cache Entries */
274     struct dce_s        *ips_dce_default;
275     uint_t              ips_dce_hashsize;
276     struct dcb_s        *ips_dce_hash_v4;
277     struct dcb_s        *ips_dce_hash_v6;

279     /* pending binds */
280     mblk_t              *ips_ip6_asp_pending_ops;
281     mblk_t              *ips_ip6_asp_pending_ops_tail;

283     /* Synchronize updates with table usage */
284     mblk_t              *ips_ip6_asp_pending_update; /* pending table updates */

286     boolean_t           ips_ip6_asp_uip;           /* table update in progress */
287     kmutex_t            ips_ip6_asp_lock;          /* protect all the above */
288     uint32_t            ips_ip6_asp_refcnt;        /* outstanding references */

290     struct ip6_asp      *ips_ip6_asp_table;
291     /* The number of policy entries in the table */
292     uint_t              ips_ip6_asp_table_count;

294     struct conn_s       *ips_ip_g_mrouter;

296     /* Time since last icmp_pkt_err */
297     clock_t             ips_icmp_pkt_err_last;
298     /* Number of packets sent in burst */
299     uint_t              ips_icmp_pkt_err_sent;

301     /* Protected by ip_mi_lock */
302     void                *ips_ip_g_head; /* IP Instance Data List Head */
303     void                *ips_arp_g_head; /* ARP Instance Data List Head */

305     /* Multirouting stuff */
306     /* Interval (in ms) between consecutive 'bad MTU' warnings */
307     hrtime_t            ips_ip_multirt_log_interval;
308     /* Time since last warning issued. */
309     hrtime_t            ips_multirt_bad_mtu_last_time;

311     /*
312     * CGTP hooks. Enabling and disabling of hooks is controlled by an
313     * IP tunable 'ips_ip_cgtp_filter'.
314     */
315     struct cgtp_filter_ops *ips_ip_cgtp_filter_ops;

317     struct ipsq_s       *ips_ipsq_g_head;
318     uint_t              ips_ill_index; /* Used to assign interface indicies */
319     /* When set search for unused index */
320     boolean_t           ips_ill_index_wrap;

322     uint_t              ips_loopback_packets;

324     /* NDP/NCE structures for IPv4 and IPv6 */
325     struct ndp_g_s      *ips_ndp4;
326     struct ndp_g_s      *ips_ndp6;

328     /* ip_mroute stuff */
329     kmutex_t            ips_ip_g_mrouter_mutex;

331     struct mrtstat      *ips_mrtstat; /* Stats for netstat */
332     int                 ips_saved_ip_forwarding;

```

```

334     /* numvifs is only a hint about the max interface being used. */
335     ushort_t           ips_numvifs;
336     kmutex_t           ips_numvifs_mutex;

338     struct vif          *ips_vifs;
339     struct mfc_b        *ips_mfcs; /* kernel routing table */
340     struct tbf          *ips_tbf;
341     /*
342     * One-back cache used to locate a tunnel's vif,
343     * given a datagram's src ip address.
344     */
345     ipaddr_t            ips_last_encap_src;
346     struct vif          *ips_last_encap_vif;
347     kmutex_t            ips_last_encap_lock; /* Protects the above */

349     /*
350     * reg_vif_num is protected by numvifs_mutex
351     */
352     /* Whether or not special PIM assert processing is enabled. */
353     ushort_t           ips_reg_vif_num; /* Index to Register vif */
354     int                 ips_pim_assert;

356     union ill_g_head_u *ips_ill_g_heads; /* ILL List Head */

358     kstat_t             *ips_loopback_ksp;

360     /* Array of conn drain lists */
361     struct idl_tx_list_s *ips_idl_tx_list;
362     uint_t              ips_conn_drain_list_cnt; /* Count of conn_drain_list */

364     /*
365     * ID used to assign next free one.
366     * Increases by one. Once it wraps we search for an unused ID.
367     */
368     uint_t              ips_ip_src_id;
369     boolean_t           ips_srcid_wrapped;

371     struct srcid_map    *ips_srcid_head;
372     krmlock_t           ips_srcid_lock;

374     uint64_t            ips_ipif_g_seqid; /* Used only for sctp_addr.c */
375     union phyint_list_u *ips_phyint_g_list; /* start of phyint list */

377     /* ip_netinfo.c */
378     hook_family_t       ips_ipv4root;
379     hook_family_t       ips_ipv6root;
380     hook_family_t       ips_arproot;

382     net_handle_t        ips_ipv4_net_data;
383     net_handle_t        ips_ipv6_net_data;
384     net_handle_t        ips_arp_net_data;

386     /*
387     * Hooks for firewalling
388     */
389     hook_event_t        ips_ip4_physical_in_event;
390     hook_event_t        ips_ip4_physical_out_event;
391     hook_event_t        ips_ip4_forwarding_event;
392     hook_event_t        ips_ip4_loopback_in_event;
393     hook_event_t        ips_ip4_loopback_out_event;

395     hook_event_t        ips_ip6_physical_in_event;
396     hook_event_t        ips_ip6_physical_out_event;
397     hook_event_t        ips_ip6_forwarding_event;
398     hook_event_t        ips_ip6_loopback_in_event;
399     hook_event_t        ips_ip6_loopback_out_event;

```

```

401     hook_event_t      ips_arp_physical_in_event;
402     hook_event_t      ips_arp_physical_out_event;
403     hook_event_t      ips_arp_nic_events;

405     hook_event_token_t ips_ipv4firewall_physical_in;
406     hook_event_token_t ips_ipv4firewall_physical_out;
407     hook_event_token_t ips_ipv4firewall_forwarding;
408     hook_event_token_t ips_ipv4firewall_loopback_in;
409     hook_event_token_t ips_ipv4firewall_loopback_out;

411     hook_event_token_t ips_ipv6firewall_physical_in;
412     hook_event_token_t ips_ipv6firewall_physical_out;
413     hook_event_token_t ips_ipv6firewall_forwarding;
414     hook_event_token_t ips_ipv6firewall_loopback_in;
415     hook_event_token_t ips_ipv6firewall_loopback_out;

417     hook_event_t      ips_ip4_nic_events;
418     hook_event_t      ips_ip6_nic_events;
419     hook_event_token_t ips_ipv4nicevents;
420     hook_event_token_t ips_ipv6nicevents;

422     hook_event_token_t ips_arp_physical_in;
423     hook_event_token_t ips_arp_physical_out;
424     hook_event_token_t ips_arpnicevents;

426     net_handle_t      ips_ip4_observe_pr;
427     net_handle_t      ips_ip6_observe_pr;
428     hook_event_t      ips_ip4_observe;
429     hook_event_t      ips_ip6_observe;
430     hook_event_token_t ips_ipv4observing;
431     hook_event_token_t ips_ipv6observing;

433     struct __ldi_ident *ips_ldi_ident;

435 /* ipmp.c */
436     krwlock_t          ips_ipmp_lock;
437     mod_hash_t         *ips_ipmp_grp_hash;

439 };
440 typedef struct ip_stack ip_stack_t;

442 /* Finding an ip_stack_t */
443 #define CONNQ_TO_IPST(_q)      (Q_TO_CONN(_q)->conn_netstack->netstack_ip)
444 #define ILLQ_TO_IPST(_q)      (((ill_t *)(_q)->q_ptr)->ill_ipst)
445 #define PHYINT_TO_IPST(phyi)  ((phyi)->phyint_ipsq->ipsq_ipst)

447 #else /* _KERNEL */
448 typedef int ip_stack_t;
449 #endif /* _KERNEL */

451 #ifdef __cplusplus
452 }
453 #endif

455 #endif /* _INET_IP_STACK_H */

```

new/usr/src/uts/common/inet/ipclassifier.h

1

```
*****
27289 Sat Aug 18 10:37:18 2012
new/usr/src/uts/common/inet/ipclassifier.h
dccp: conn_t
*****

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/dccp.h>
38 #endif /* ! codereview */
39 #include <inet/ip6.h>
40 #include <netinet/in.h> /* for IPPROTO_* constants */
41 #include <sys/sdt.h>
42 #include <sys/socket_proto.h>
43 #include <sys/sunddi.h>
44 #include <sys/sunldi.h>

46 typedef void (*edesc_rpf)(void *, mblk_t *, void *, ip_recv_attr_t *);
47 struct icmph_s;
48 struct icmp6_hdr;
49 typedef boolean_t (*edesc_vpf)(conn_t *, void *, struct icmph_s *,
50 struct icmp6_hdr *, ip_recv_attr_t *);

52 /*
53 * =====
54 * = The CONNECTION =
55 * =====
56 */

58 /*
59 * The connection structure contains the common information/flags/ref needed.
60 * Implementation will keep the connection struct, the layers (with their
61 * respective data for event i.e. tcp_t if event was tcp_input_data) all in one
```

new/usr/src/uts/common/inet/ipclassifier.h

2

```
62 * contiguous memory location.
63 */

65 /* Conn Flags */
66 /* Unused 0x00020000 */
67 /* Unused 0x00040000 */
68 #define IPCL_FULLY_BOUND 0x00080000 /* Bound to correct queue */
69 /* Unused 0x00100000 */
70 /* Unused 0x00200000 */
71 /* Unused 0x00400000 */
72 #define IPCL_CL_LISTENER 0x00800000 /* Cluster listener */
73 /* Unused 0x01000000 */
74 /* Unused 0x02000000 */
75 /* Unused 0x04000000 */
76 /* Unused 0x08000000 */
77 /* Unused 0x10000000 */
78 /* Unused 0x20000000 */
79 #define IPCL_CONNECTED 0x40000000 /* Conn in connected table */
80 #define IPCL_BOUND 0x80000000 /* Conn in bind table */

82 /* Flags identifying the type of conn */
83 #define IPCL_TCPCONN 0x00000001 /* From tcp_conn_cache */
84 #define IPCL_SCTPCONN 0x00000002 /* From sctp_conn_cache */
85 #define IPCL_IPCCONN 0x00000004 /* From ip_conn_cache */
86 #define IPCL_UDPCONN 0x00000008 /* From udp_conn_cache */
87 #define IPCL_RAWIPCONN 0x00000010 /* From rawip_conn_cache */
88 #define IPCL_RTSCONN 0x00000020 /* From rts_conn_cache */
89 #define IPCL_DCCPCONN 0x00000040 /* From dccp_conn_cache */
90 /* Unused 0x00000040 */
91 #define IPCL_IPTUN 0x00000080 /* iptun module above us */

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_mininfo_dev;      /* Device */
143     vmem_t    *ip_mininfo_arena;   /* Arena */
144 } ip_helper_mininfo_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_mininfo_t *iphs_mininfo;
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
189             crbb_recvttl : 1,          /* IP_RECVTTL option */
190             crbb_ip_recvpktinfo : 1,   /* IP*_RECVPKTINFO option */
191             crbb_ipv6_recvhoplimit : 1, /* IPV6_RECVHOPLIMIT option */
192             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.crbb.crbb_recvdstaddr
209 #define crb_recvopts  crbu.crbb.crbb_recvopts
210 #define crb_recvif    crbu.crbb.crbb_recvif
211 #define crb_recvslla  crbu.crbb.crbb_recvslla
212 #define crb_recvttl   crbu.crbb.crbb_recvttl
213 #define crb_ip_recvpktinfo crbu.crbb.crbb_ip_recvpktinfo
214 #define crb_ipv6_recvhoplimit crbu.crbb.crbb_ipv6_recvhoplimit
215 #define crb_ipv6_recvhopopts crbu.crbb.crbb_ipv6_recvhopopts
216 #define crb_ipv6_recvdstopts crbu.crbb.crbb_ipv6_recvdstopts
217 #define crb_ipv6_recvrthdr crbu.crbb.crbb_ipv6_recvrthdr
218 #define crb_old_ipv6_recvdstopts crbu.crbb.crbb_old_ipv6_recvdstopts
219 #define crb_ipv6_recvrthdrdstopts crbu.crbb.crbb_ipv6_recvrthdrdstopts
220 #define crb_ipv6_recvtclass crbu.crbb.crbb_ipv6_recvtclass
221 #define crb_recvucred  crbu.crbb.crbb_recvucred
222 #define crb_timestamp crbu.crbb.crbb_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 */

239     union {
240         tcp_t      *cp_tcp;      /* Pointer to the tcp struct */
241         struct udp_s *cp_udp;    /* Pointer to the udp struct */
242         struct icmp_s *cp_icmp; /* Pointer to rawip struct */
243         struct rts_s *cp_rts;   /* Pointer to rts struct */
244         struct iptun_s *cp_iptun; /* Pointer to iptun_t */
245         struct sctp_s *cp_sctp; /* For IPCL_SCTPCONN */
246         struct dccp_s *cp_dccp; /* Pointer to dccp struct */

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_rcv;           /* Pointer to rcv routine */
264     edesc_rpf        conn_rcvicmp;      /* 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_rcv_ancillary

272     /* Options for receive-side ancillary data */
273     crb_t            conn_rcv_ancillary;

275     squeue_t         *conn_sq;           /* 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_sq : 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_keepalive : 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_OOBINLINE 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_sq;     /* Squeue at open time */
311     squeue_t         *conn_final_sq;       /* 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     ilg_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 sockfs */
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 #ifdef 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 queue 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  * IPCL_PROTO_MATCH() and IPCL_PROTO_MATCH_V6() only matches conns with
557  * the specified ira_zoneid or conn_allzones by calling conn_wantpacket.
558  */
559 #define IPCL_PROTO_MATCH(connp, ira, ipha) \
560     (((connp)->conn_laddr_v4 == INADDR_ANY) || \
561      (((connp)->conn_laddr_v4 == ((ipha)->ipha_dst)) && \
562       (((connp)->conn_faddr_v4 == INADDR_ANY) || \
563        ((connp)->conn_faddr_v4 == ((ipha)->ipha_src)))) && \
564      conn_wantpacket((connp), (ira), (ipha)))
565
566 #define IPCL_PROTO_MATCH_V6(connp, ira, ip6h) \
567     ((IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
568      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &((ip6h)->ip6_dst)) && \
569      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
570      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &((ip6h)->ip6_src)))))) && \
571      (conn_wantpacket_v6((connp), (ira), (ip6h))))
572
573 #define IPCL_CONN_HASH(src, ports, ipst) \
574     ((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^ \
575      ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_conn_fanout_size)
576
577 #define IPCL_CONN_HASH_V6(src, ports, ipst) \
578     IPCL_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))
579
580 #define IPCL_CONN_MATCH(connp, proto, src, dst, ports) \
581     (((connp)->conn_proto == (proto) && \
582      (connp)->conn_ports == (ports) && \
583      _IPCL_V4_MATCH((connp)->conn_faddr_v6, (src)) && \
584      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (dst)) && \
585      !((connp)->conn_ipv6_v6only))
586
587 #define IPCL_CONN_MATCH_V6(connp, proto, src, dst, ports) \

```

```

588     (((connp)->conn_proto == (proto) && \
589      (connp)->conn_ports == (ports) && \
590      IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(src)) && \
591      IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(dst))))
592
593 #define IPCL_PORT_HASH(port, size) \
594     (((port) >> 8) ^ (port)) & ((size) - 1)
595
596 #define IPCL_BIND_HASH(lport, ipst) \
597     ((unsigned)((lport) >> 8) ^ (lport)) % \
598     (ipst)->ips_ipcl_bind_fanout_size)
599
600 #define IPCL_BIND_MATCH(connp, proto, laddr, lport) \
601     (((connp)->conn_proto == (proto) && \
602      (connp)->conn_lport == (lport) && \
603      (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
604      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) && \
605      !((connp)->conn_ipv6_v6only))
606
607 #define IPCL_BIND_MATCH_V6(connp, proto, laddr, lport) \
608     (((connp)->conn_proto == (proto) && \
609      (connp)->conn_lport == (lport) && \
610      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) || \
611      IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6)))
612
613 /*
614  * We compare conn_laddr since it captures both connected and a bind to
615  * a multicast or broadcast address.
616  * The caller needs to match the zoneid and also call conn_wantpacket
617  * for multicast, broadcast, or when conn_incoming_ifindex is set.
618  */
619 #define IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr) \
620     (((connp)->conn_lport == (lport)) && \
621      (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
622      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) && \
623      _IPCL_V4_MATCH_ANY((connp)->conn_faddr_v6) || \
624      _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)) && \
625      (connp)->conn_fport == (fport)))) && \
626      !((connp)->conn_ipv6_v6only))
627
628 /*
629  * We compare conn_laddr since it captures both connected and a bind to
630  * a multicast or broadcast address.
631  * The caller needs to match the zoneid and also call conn_wantpacket_v6
632  * for multicast or when conn_incoming_ifindex is set.
633  */
634 #define IPCL_UDP_MATCH_V6(connp, lport, laddr, fport, faddr) \
635     (((connp)->conn_lport == (lport)) && \
636      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
637      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) && \
638      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
639      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(faddr)) && \
640      (connp)->conn_fport == (fport))))))
641
642 #define IPCL_IPTUN_HASH(laddr, faddr) \
643     ((ntohl(laddr) ^ ((ntohl(faddr) << 24) | (ntohl(faddr) >> 8))) % \
644      ipcl_iptun_fanout_size)
645
646 #define IPCL_IPTUN_HASH_V6(laddr, faddr) \
647     (IPCL_IPTUN_HASH((laddr)->s6_addr32[0] ^ (laddr)->s6_addr32[1] ^ \
648      (faddr)->s6_addr32[2] ^ (faddr)->s6_addr32[3], \
649      (faddr)->s6_addr32[0] ^ (faddr)->s6_addr32[1] ^ \
650      (laddr)->s6_addr32[2] ^ (laddr)->s6_addr32[3]))
651
652 #define IPCL_IPTUN_MATCH(connp, laddr, faddr) \
653     (_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_CONN_HASH(src, ports, ipst) \
665     ((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^ \
666     ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_dccp_conn_fanout_size)

668 #define IPCL_DCCP_CONN_HASH_V6(src, ports, ipst) \
669     IPCL_DCCP_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))

671 #define IPCL_DCCP_BIND_HASH(lport, ipst) \
672     ((unsigned)((lport) >> 8) ^ (lport)) % \
673     (ipst)->ips_ipcl_dccp_bind_fanout_size)

676 #endif /* ! codereview */
677 #define CONN_G_HASH_SIZE      1024

679 /* Raw socket hash function. */
680 #define IPCL_RAW_HASH(lport, ipst) \
681     IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_raw_fanout_size)

683 /*
684  * This is similar to IPCL_BIND_MATCH except that the local port check
685  * is changed to a wildcard port check.
686  * We compare conn_laddr since it captures both connected and a bind to
687  * a multicast or broadcast address.
688  */
689 #define IPCL_RAW_MATCH(connp, proto, laddr) \
690     ((connp)->conn_proto == (proto) && \
691     (connp)->conn_lport == 0 && \
692     (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
693     _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))))

695 #define IPCL_RAW_MATCH_V6(connp, proto, laddr) \
696     ((connp)->conn_proto == (proto) && \
697     (connp)->conn_lport == 0 && \
698     (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
699     IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr))))

701 /* Function prototypes */
702 extern void ipcl_g_init(void);
703 extern void ipcl_init(ip_stack_t *);
704 extern void ipcl_g_destroy(void);
705 extern void ipcl_destroy(ip_stack_t *);
706 extern conn_t *ipcl_conn_create(uint32_t, int, netstack_t *);
707 extern void ipcl_conn_destroy(conn_t *);

709 void ipcl_hash_insert_wildcard(connf_t *, conn_t *);
710 void ipcl_hash_remove(conn_t *);
711 void ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp);

713 extern int ipcl_bind_insert(conn_t *);
714 extern int ipcl_bind_insert_v4(conn_t *);
715 extern int ipcl_bind_insert_v6(conn_t *);
716 extern int ipcl_conn_insert(conn_t *);
717 extern int ipcl_conn_insert_v4(conn_t *);
718 extern int ipcl_conn_insert_v6(conn_t *);
719 extern conn_t *ipcl_get_next_conn(connf_t *, conn_t *, uint32_t);

```

```

721 conn_t *ipcl_classify_v4(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
722     ip_stack_t *);
723 conn_t *ipcl_classify_v6(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
724     ip_stack_t *);
725 conn_t *ipcl_classify(mblk_t *, ip_recv_attr_t *, ip_stack_t *);
726 conn_t *ipcl_classify_raw(mblk_t *, uint8_t, uint32_t, ipha_t *,
727     ip6_t *, ip_recv_attr_t *, ip_stack_t *);
728 conn_t *ipcl_iptun_classify_v4(ipaddr_t *, ipaddr_t *, ip_stack_t *);
729 conn_t *ipcl_iptun_classify_v6(in6_addr_t *, in6_addr_t *, ip_stack_t *);
730 void ipcl_globalhash_insert(conn_t *);
731 void ipcl_globalhash_remove(conn_t *);
732 void ipcl_walk(pfv_t, void *, ip_stack_t *);
733 conn_t *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack_t *);
734 conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
735     ip_stack_t *);
736 /*
737 conn_t *ipcl_dccp_lookup_reversed_ipv4(ipha_t *, dccpha_t *, int, ip_stack_t *);
738 conn_t *ipcl_dccp_lookup_reversed_ipv6(ip6_t *, dccpha_t *, int, uint_t,
739     ip_stack_t *);
740 */
741 #endif /* ! codereview */
742 conn_t *ipcl_lookup_listener_v4(uint16_t, ipaddr_t, zoneid_t, ip_stack_t *);
743 conn_t *ipcl_lookup_listener_v6(uint16_t, in6_addr_t *, uint_t, zoneid_t,
744     ip_stack_t *);
745 int conn_trace_ref(conn_t *);
746 int conn_untrace_ref(conn_t *);
747 void ipcl_conn_cleanup(conn_t *);
748 extern uint_t conn_recvancillary_size(conn_t *, crb_t, ip_recv_attr_t *,
749     mblk_t *, ip_pkt_t *);
750 extern void conn_recvancillary_add(conn_t *, crb_t, ip_recv_attr_t *,
751     ip_pkt_t *, uchar_t *, uint_t);
752 conn_t *ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *, ipha_t *, tcpha_t *,
753     ip_stack_t *);
754 conn_t *ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *, ip6_t *, tcpha_t *,
755     ip_stack_t *);

757 extern int ip_create_helper_stream(conn_t *, ldi_ident_t);
758 extern void ip_free_helper_stream(conn_t *);
759 extern int ip_helper_stream_setup(queue_t *, dev_t *, int, int,
760     cred_t *, boolean_t);

762 #ifdef __cplusplus
763 }
764 #endif

766 #endif /* _INET_IPCLASSIFIER_H */

```

new/usr/src/uts/common/inet/kstatcom.h

1

12863 Sat Aug 18 10:37:18 2012

new/usr/src/uts/common/inet/kstatcom.h

dccp: stats

_____unchanged_portion_omitted_

```
470 typedef struct dccp_named_kstat {
471     kstat_named_t activeOpens;
472     kstat_named_t passiveOpens;
473     kstat_named_t inSegs;
474     kstat_named_t outSegs;
475 } dccp_named_kstat_t;
476 #endif /* ! codereview */

478 #define NUM_OF_FIELDS(S)      (sizeof (S) / sizeof (kstat_named_t))

480 #ifdef __cplusplus
481 }
482 #endif

484 #endif /* _INET_KSTATCOM_H */
```

new/usr/src/uts/common/inet/mib2.h

1

```
*****
63821 Sat Aug 18 10:37:18 2012
new/usr/src/uts/common/inet/mib2.h
dccc: MIB-II
*****
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]etsockopt() calls. get uses T_CURRENT, set uses T_NEOGTIATE
47  * MGMT_flags value. The following definition of opthdr is taken from
48  * socket.h:
49  *
50  * An option specification consists of an opthdr, followed by the value of
51  * the option. An options buffer contains one or more options. The len
52  * field of opthdr 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 opthdr {
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 #define MIB2_DCCP6 (IPPROTO_MAX+18)
101 #endif /* ! codereview */

103 /*
104  * Define range of levels for use with MIB2.*
105 */
106 #define MIB2_RANGE_START (IPPROTO_MAX+1)
107 #define MIB2_RANGE_END (IPPROTO_MAX+18)
99 #define MIB2_RANGE_END (IPPROTO_MAX+16)

110 #define EXPER 1024 /* experimental - not part of mib */
111 #define EXPER_IGMP (EXPER+1)
112 #define EXPER_DVMRP (EXPER+2)
113 #define EXPER_RAWIP (EXPER+3)
114 #define EXPER_IP_AND_ALL_IRES (EXPER+4)

116 /*
117  * Define range of levels for experimental use
118 */
119 #define EXPER_RANGE_START (EXPER+1)
120 #define EXPER_RANGE_END (EXPER+4)

122 #define BUMP_MIB(s, x) { \
123     extern void __dtrace_probe__mib_##x(int, void *); \
124     void *stataddr = &((s)->x); \
125     __dtrace_probe__mib_##x(1, stataddr); \
126     (s)->x++; \

```

```

127 }
    unchanged_portion_omitted_

1788 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1789 #pragma pack()
1790 #endif

1792 /*
1793  * the DCCP group
1794  */
1795 #define MIB2_DCCP_CONN          18
1796 #define MIB2_DCCP6_CONN        19

1798 #define MIB2_DCCP_closed        1
1799 #define MIB2_DCCP_listen        2

1801 /* Pack data to make struct size the same for 32- and 64-bits */
1802 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1803 #pragma pack(4)
1804 #endif

1806 typedef struct mib2_dccp {

1808     /* # of direct transitions CLOSED -> ACK-SENT { dccp 5 } */
1809     Counter dccpActiveOpens;
1810     /* # of direct transitions LISTEN -> ACK-RCVD { dccp 6 } */
1811     Counter dccpPassiveOpens;
1812     /* # of direct SIN-SENT/RCVD -> CLOSED/LISTEN { dccp 7 } */
1813     Counter dccpAttemptFails;
1814     /* # of direct ESTABLISHED/CLOSE-WAIT -> CLOSED { dccp 8 } */
1815     Counter dccpEstabResets;
1816     /* # of connections ESTABLISHED or CLOSE-WAIT { dccp 9 } */
1817     Gauge dccpCurrEstab;
1818     /* total # of segments rcv'd { dccp 10 } */
1819     Counter dccpInSegs;
1820     /* total # of segments sent { dccp 11 } */
1821     Counter dccpOutSegs;
1822     /* total # of segments retransmitted { dccp 12 } */
1823     Counter dccpRetransSegs;

1825     Counter dccpTimRetrans;
1826     /* total # of retransmit timeouts dropping the connection */
1827     Counter dccpTimRetransDrop;
1828     /* total # of keepalive timeouts */
1829     Counter dccpTimKeepalive;
1830     /* total # of keepalive timeouts sending a probe */
1831     Counter dccpTimKeepaliveProbe;
1832     /* total # of keepalive timeouts dropping the connection */
1833     Counter dccpTimKeepaliveDrop;

1835     Counter dccpOutDataSegs;
1836     Counter dccpOutDataBytes;

1838     int dccpEntrySize;
1839     int dccp6EntrySize;

1841     int dccpConnTableSize;
1842     int dccp6ConnTableSize;

1844     Counter64 dccpHCInDatagrams;
1845     /* total # of segments rcv'd { tcp 17 } */
1846     Counter64 dccpHCInSegs;
1847     /* total # of segments sent { tcp 18 } */
1848     Counter64 dccpHCOutSegs;
1849 } mib2_dccp_t;
1850 #define MIB_FIRST_NEW_ELM_mib2_dccp_t dccpHCInDatagrams

```

```

1852 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1853 #pragma pack()
1854 #endif

1856 /* Pack data to make struct size the same for 32- and 64-bits */
1857 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1858 #pragma pack(4)
1859 #endif

1861 typedef struct mib2_dccpConnEntry {
1862     int dccpConnState;
1863     IpAddress dccpConnLocalAddress;
1864     int dccpConnLocalPort;
1865     IpAddress dccpConnRemAddress;
1866     int dccpConnRemPort;

1868     struct dccpConnEntryInfo_s {
1869         /* current rto (retransmit timeout) */
1870         Gauge ce_rto;
1871         /* current max segment size */
1872         Gauge ce_mss;
1873         /* actual internal state */
1874         int ce_state;
1875     } dccpConnEntryInfo;

1877     uint32_t dccpConnCreationProcess;
1878     uint64_t dccpConnCreationTime;
1879 } mib2_dccpConnEntry_t;
1880 #define MIB_FIRST_NEW_ELM_mib2_dccpConnEntry_t dccpConnCreationProcess

1882 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1883 #pragma pack()
1884 #endif

1886 /* Pack data to make struct size the same for 32- and 64-bits */
1887 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1888 #pragma pack(4)
1889 #endif

1891 typedef struct mib2_dccp6ConnEntry {
1892     Ip6Address dccp6ConnLocalAddress;
1893     int dccp6ConnLocalPort;
1894     Ip6Address dccp6ConnRemAddress;
1895     int dccp6ConnRemPort;
1896     /* Interface index or zero { ipv6DccpConnEntry 5 } */
1897     DeviceIndex dccp6ConnIfIndex;
1898     /* State of dccp6 connection { ipv6DccpConnEntry 6 } RW */
1899     int dccp6ConnState;

1901     struct dccp6ConnEntryInfo_s {
1902         /* current rto (retransmit timeout) */
1903         Gauge ce_rto;
1904         /* current max segment size */
1905         Gauge ce_mss;
1906         /* actual internal state */
1907         int ce_state;
1908     } dccp6ConnEntryInfo;

1910     /* PID of the processes that created this connection */
1911     uint32_t dccp6ConnCreationProcess;
1912     /* System uptime when the connection was created */
1913     uint64_t dccp6ConnCreationTime;
1914 } mib2_dccp6ConnEntry_t;
1915 #define MIB_FIRST_NEW_ELM_mib2_dccp6ConnEntry_t dccp6ConnCreationProcess

```

new/usr/src/uts/common/inet/mib2.h

5

```
1917 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1918 #pragma pack()
1919 #endif
1920 #endif /* ! codereview */

1922 #ifdef __cplusplus
1923 }
1924 #endif

1926 #endif /* _INET_MIB2_H */
```


new/usr/src/uts/common/inet/sctp/sctp_impl.h

1

```
*****
46352 Sat Aug 18 10:37:19 2012
new/usr/src/uts/common/inet/sctp/sctp_impl.h
sctp: align to cache line
*****
_____unchanged_portion_omitted_____

428 /*
429  * Bind hash array size and hash function. The size must be a power
430  * of 2 and lport must be in host byte order.
431  */
432 #define SCTP_BIND_FANOUT_SIZE 2048
433 #define SCTP_BIND_HASH(lport) (((lport) * 31) & (SCTP_BIND_FANOUT_SIZE - 1))

435 /* options that SCTP negotiates during association establishment */
436 #define SCTP_PRSTCTP_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 {
```

new/usr/src/uts/common/inet/sctp/sctp_impl.h

2

```
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_pms;
564     uint32_t      sf_cwnd;
565     uint32_t      sf_ssthresh;
566     uint32_t      sf_suna;      /* sent - unack'ed */
567     uint32_t      sf_pba;      /* partial bytes acked */
568     uint32_t      sf_acked;
569     int64_t       sf_lastactive;
570     mblk_t        *sf_timer_mp; /* retransmission timer control */
571     uint32_t

572         sf_hb_pending : 1,
573         sf_timer_running : 1,
574         sf_df : 1,
575         sf_pmtu_discovered : 1,

577         sf_rc_timer_running : 1,
578         sf_isv4 : 1,
579         sf_hb_enabled : 1;

581     mblk_t        *sf_rc_timer_mp; /* reliable control chunk timer */
582     ip_xmit_attr_t *sf_ixa;        /* Transmit attributes */
583     uint32_t      sf_T3expire;     /* # of times T3 timer expired */

585     uint64_t      sf_hb_secret; /* per addr "secret" in heartbeat */
586     uint32_t      sf_rxt_unacked; /* # unack'ed retransmitted bytes */
587 } sctp_faddr_t;

589 /* Flags to indicate supported address type in the PARM_SUP_ADDRS. */
590 #define PARM_SUPP_V6    0x1
591 #define PARM_SUPP_V4    0x2

593 /*
594  * Set heartbeat interval plus jitter. The jitter is supposed to be random,
595  * up to +/- 50% of the RTO. We use gethrtime() here for performance reason
596  * as the jitter does not really need to be "very" random.
597  */
598 #define SET_HB_INTVL(fp) \
599     ((fp)->sf_hb_interval + (fp)->sf_rto + ((fp)->sf_rto >> 1) - \
600     (uint_t)gethrtime() % (fp)->sf_rto)

602 #define Sctp_IPIF_HASH 16

604 typedef struct sctp_ipif_hash_s {
605     list_t      sctp_ipif_list;
606     int         ipif_count;
607     krwlock_t   ipif_hash_lock;
608 } sctp_ipif_hash_t;

611 /*
612  * Initialize cwnd according to RFC 3390. def_max_init_cwnd is
613  * either sctp_slow_start_initial or sctp_slow_start_after_idle
614  * depending on the caller.
615  */
616 #define SET_CWND(fp, mss, def_max_init_cwnd) \
617 { \
618     (fp)->sf_cwnd = MIN(def_max_init_cwnd * (mss), \

```

```

619         MIN(4 * (mss), MAX(2 * (mss), 4380 / (mss) * (mss)))); \
620 }

623 struct sctp_s;

625 /*
626  * Control structure for each open SCTP stream,
627  * defined only within the kernel or for a kmem user.
628  * NOTE: sctp_reinit_values MUST have a line for each field in this structure!
629  */
630 #if (defined(_KERNEL) || defined(_KMEMUSER))

632 typedef struct sctp_s {

634     /*
635      * The following is shared with (and duplicated) in IP, so if you
636      * make changes, make sure you also change things in ip_sctp.c.
637      */
638     struct sctp_s *sctp_conn_hash_next;
639     struct sctp_s *sctp_conn_hash_prev;

641     struct sctp_s *sctp_listen_hash_next;
642     struct sctp_s *sctp_listen_hash_prev;

644     sctp_tf_t      *sctp_listen_tfp; /* Ptr to tf */
645     sctp_tf_t      *sctp_conn_tfp;  /* Ptr to tf */

647     /* Global list of sctp */
648     list_node_t    sctp_list;

650     sctp_faddr_t    *sctp_faddrs;
651     int              sctp_nfaddrs;
652     sctp_ipif_hash_t sctp_saddrs[SCTP_IPIF_HASH];
653     int              sctp_nsaddrs;

655     kmutex_t        sctp_lock;
656     kcondvar_t      sctp_cv;
657     boolean_t       sctp_running;

659 #define sctp_ulpd      sctp_connp->conn_upper_handle
660 #define sctp_upcalls   sctp_connp->conn_upcalls

662 #define sctp_ulp_newconn      sctp_upcalls->su_newconn
663 #define sctp_ulp_connected   sctp_upcalls->su_connected
664 #define sctp_ulp_disconnected sctp_upcalls->su_disconnected
665 #define sctp_ulp_opctl       sctp_upcalls->su_opctl
666 #define sctp_ulp_rcv         sctp_upcalls->su_rcv
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_ostrcnts;

699     mblk_t      *sctp_pad_mp;      /* pad unaligned data chunks */

701     /* sendmsg() default parameters */
702     uint16_t      sctp_def_stream; /* default stream id */
703     uint16_t      sctp_def_flags;  /* default xmit flags */
704     uint32_t      sctp_def_ppid;   /* default payload id */
705     uint32_t      sctp_def_context; /* default context */
706     uint32_t      sctp_def_timetolive; /* default msg TTL */

708     /* Inbound data tracking */
709     sctp_set_t     *sctp_sack_info; /* Sack tracking */
710     mblk_t         *sctp_ack_mp;    /* Delayed ACK timer block */
711     sctp_instr_t   *sctp_instr;     /* Instream trackers */
712     mblk_t         *sctp_uo_frags;  /* Un-ordered msg. fragments */
713     uint32_t       sctp_ftsn;       /* Peer's TSN */
714     uint32_t       sctp_lastacked;  /* last cumtsn SACKd */
715     uint16_t       sctp_num_istr;   /* No. of instreams */
716     int32_t        sctp_istr_nmsgs; /* No. of chunks in instreams */
717     int32_t        sctp_sack_gaps;  /* No. of received gaps */
718     int32_t        sctp_sack_toggle; /* SACK every other pkt */

720     /* RTT calculation */
721     uint32_t       sctp_rtt_tsn;
722     int64_t        sctp_out_time;

724     /* Stats can be reset by snmp users kstat, netstat and snmp agents */
725     uint64_t       sctp_opkts;      /* sent pkts */
726     uint64_t       sctp_obchunks;   /* sent control chunks */
727     uint64_t       sctp_odchunks;   /* sent ordered data chunks */
728     uint64_t       sctp_oudchunks;  /* sent unord data chunks */
729     uint64_t       sctp_rxtchunks;  /* retransmitted chunks */
730     uint64_t       sctp_ipkts;      /* recv pkts */
731     uint64_t       sctp_ibchunks;   /* recv control chunks */
732     uint64_t       sctp_idchunks;   /* recv ordered data chunks */
733     uint64_t       sctp_iudchunks;  /* recv unord data chunks */
734     uint64_t       sctp_fragdmsgs;
735     uint64_t       sctp_reassmsgs;
736     uint32_t       sctp_Tlexpire;    /* # 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,
818
819         sctp_rcvpeererr : 1,
820         sctp_rcvshutdownevnt : 1,
821         sctp_rcvpdevnt : 1,
822         sctp_rcvvaldevnt : 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_flowctrlld sctp_bits.sctp_flowctrlld
848
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_rcvpdevnt sctp_events.sctp_rcvpdevnt
856 #define sctp_rcvvaldevnt sctp_events.sctp_rcvvaldevnt
857
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;
868
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 */
873
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;
878
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 */
885
886 mblk_t *sctp_heartbeat_mp; /* Timer block for heartbeats */
887 uint32_t sctp_hb_interval; /* Default hb_interval */
888
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 */
893
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 */
898
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;
904
905 /* IPv6 ancillary data */
906 uint_t sctp_rcvifindex; /* last rcvd IPV6_RCVPKTINFO */
907 uint_t sctp_rcvhops; /* " IPV6_RECVOPLIMIT */
908 uint_t sctp_rcvclass; /* " IPV6_RECVCCLASS */
909 ip6_hbh_t *sctp_hopopts; /* " IPV6_RECVOPOPTS */
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;
917
918 /* Stats */
919 uint64_t sctp_msgcount;
920 uint64_t sctp_prsctpdrop;
921
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 */
926
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 */
930
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_recvg(sctp_t *, mblk_t *, boolean_t, \
985         ip_recv_attr_t *);
986 extern void      sctp_add_unrec_parm(sctp_parm_hdr_t *, mblk_t **, boolean_t);
987 extern size_t    sctp_addr_params(sctp_t *, int, uchar_t *, boolean_t);
988 extern mblk_t    *sctp_add_proto_hdr(sctp_t *, sctp_faddr_t *, mblk_t *, int, \
989         int *);
990 extern void      sctp_addr_req(sctp_t *, mblk_t *);
991 extern sctp_t    *sctp_addrlist2sctp(mblk_t *, sctp_hdr_t *, sctp_chunk_hdr_t *, \
992         zoneid_t, sctp_stack_t *);
993 extern void      sctp_check_adv_ack_pt(sctp_t *, mblk_t *, mblk_t *);
994 extern void      sctp_assoc_event(sctp_t *, uint16_t, uint16_t, \
995         sctp_chunk_hdr_t *);

997 extern void      sctp_bind_hash_insert(sctp_tf_t *, sctp_t *, int);
998 extern void      sctp_bind_hash_remove(sctp_t *);
999 extern int      sctp_bindi(sctp_t *, in_port_t, boolean_t, int, in_port_t *);
1000 extern int      sctp_bind_add(sctp_t *, const void *, uint32_t, boolean_t, \
1001         in_port_t);
1002 extern int      sctp_bind_del(sctp_t *, const void *, uint32_t, boolean_t);
1003 extern int      sctp_build_hdrs(sctp_t *, int);

1005 extern int      sctp_check_abandoned_msg(sctp_t *, mblk_t *);
1006 extern void      sctp_clean_death(sctp_t *, int);
1007 extern void      sctp_close_eager(sctp_t *);
1008 extern int      sctp_compare_faddrsets(sctp_faddr_t *, sctp_faddr_t *);
1009 extern void      sctp_congest_reset(sctp_t *);
1010 extern void      sctp_conn_hash_insert(sctp_tf_t *, sctp_t *, int);
1011 extern void      sctp_conn_hash_remove(sctp_t *);
1012 extern void      sctp_conn_init(conn_t *);
1013 extern sctp_t    *sctp_conn_match(in6_addr_t **, uint32_t, in6_addr_t *, \
1014         uint32_t, zoneid_t, iaflags_t, sctp_stack_t *);

```

```

1015 extern void      sctp_conn_reclaim(void *);
1016 extern sctp_t    *sctp_conn_request(sctp_t *, mblk_t *, uint_t, uint_t, \
1017         sctp_init_chunk_t *, ip_recv_attr_t *);
1018 extern uint32_t  sctp_cumack(sctp_t *, uint32_t, mblk_t **);
1019 extern sctp_t    *sctp_create_eager(sctp_t *);

1021 extern void      sctp_dispatch_rput(queue_t *, sctp_t *, sctp_hdr_t *, mblk_t *, \
1022         uint_t, uint_t, in6_addr_t);
1023 extern char      *sctp_display(sctp_t *, char *);
1024 extern void      sctp_display_all(sctp_stack_t *);

1026 extern void      sctp_error_event(sctp_t *, sctp_chunk_hdr_t *, boolean_t);

1028 extern void      sctp_faddr_alive(sctp_t *, sctp_faddr_t *);
1029 extern int      sctp_faddr_dead(sctp_t *, sctp_faddr_t *, int);
1030 extern void      sctp_faddr_fini(void);
1031 extern void      sctp_faddr_init(void);
1032 extern void      sctp_fast_rexmit(sctp_t *);
1033 extern void      sctp_fill_sack(sctp_t *, unsigned char *, int);
1034 extern uint32_t  sctp_find_listener_conf(sctp_stack_t *, in_port_t);
1035 extern void      sctp_free_faddr_timers(sctp_t *);
1036 extern void      sctp_free_ftsn_set(sctp_ftsn_set_t *);
1037 extern void      sctp_free_msg(mblk_t *);
1038 extern void      sctp_free_reass(sctp_instr_t *);
1039 extern void      sctp_free_set(sctp_set_t *);
1040 extern void      sctp_ftsn_sets_fini(void);
1041 extern void      sctp_ftsn_sets_init(void);

1043 extern int      sctp_get_addrlist(sctp_t *, const void *, uint32_t *, \
1044         uchar_t **, int *, size_t *);
1045 extern int      sctp_get_addrparams(sctp_t *, sctp_t *, mblk_t *, \
1046         sctp_chunk_hdr_t *, uint_t *);
1047 extern void      sctp_get_dest(sctp_t *, sctp_faddr_t *);
1048 extern void      sctp_get_faddr_list(sctp_t *, uchar_t *, size_t);
1049 extern mblk_t    *sctp_get_first_sent(sctp_t *);
1050 extern mblk_t    *sctp_get_msg_to_send(sctp_t *, mblk_t **, mblk_t *, int *, \
1051         int32_t, uint32_t, sctp_faddr_t *);
1052 extern void      sctp_get_saddr_list(sctp_t *, uchar_t *, size_t);

1054 extern int      sctp_handle_error(sctp_t *, sctp_hdr_t *, sctp_chunk_hdr_t *, \
1055         mblk_t *, ip_recv_attr_t *);
1056 extern void      sctp_hash_destroy(sctp_stack_t *);
1057 extern void      sctp_hash_init(sctp_stack_t *);
1058 extern void      sctp_heartbeat_timer(sctp_t *);

1060 extern void      sctp_icmp_error(sctp_t *, mblk_t *);
1061 extern void      sctp_inc_taskq(sctp_stack_t *);
1062 extern void      sctp_info_req(sctp_t *, mblk_t *);
1063 extern mblk_t    *sctp_init_mp(sctp_t *, sctp_faddr_t *);
1064 extern boolean_t sctp_initialize_params(sctp_t *, sctp_init_chunk_t *, \
1065         sctp_init_chunk_t *);
1066 extern uint32_t  sctp_init2vtag(sctp_chunk_hdr_t *);
1067 extern void      sctp_intf_event(sctp_t *, in6_addr_t, int, int);
1068 extern void      sctp_input_data(sctp_t *, mblk_t *, ip_recv_attr_t *);
1069 extern void      sctp_instream_cleanup(sctp_t *, boolean_t);
1070 extern boolean_t sctp_is_a_faddr_clean(sctp_t *);

1072 extern void      *sctp_kstat_init(netstackid_t);
1073 extern void      sctp_kstat_fini(netstackid_t, kstat_t *);
1074 extern void      *sctp_kstat2_init(netstackid_t);
1075 extern void      sctp_kstat2_fini(netstackid_t, kstat_t *);

1077 extern ssize_t   sctp_link_abort(mblk_t *, uint16_t, char *, size_t, int, \
1078         boolean_t);
1079 extern void      sctp_listen_hash_insert(sctp_tf_t *, sctp_t *);
1080 extern void      sctp_listen_hash_remove(sctp_t *);

```

```

1081 extern void      sctp_listener_conf_cleanup(sctp_stack_t *);
1082 extern sctp_t     *sctp_lookup(sctp_t *, in6_addr_t *, sctp_tf_t *, uint32_t *,
1083                               int);
1084 extern sctp_faddr_t *sctp_lookup_faddr(sctp_t *, in6_addr_t *);

1086 extern mblk_t     *sctp_make_err(sctp_t *, uint16_t, void *, size_t);
1087 extern mblk_t     *sctp_make_ftsn_chunk(sctp_t *, sctp_faddr_t *,
1088                                         sctp_ftsn_set_t *, uint_t, uint32_t);
1089 extern void       sctp_make_ftsns(sctp_t *, mblk_t *, mblk_t *, mblk_t **,
1090                                   sctp_faddr_t *, uint32_t *);
1091 extern mblk_t     *sctp_make_mp(sctp_t *, sctp_faddr_t *, int);
1092 extern mblk_t     *sctp_make_sack(sctp_t *, sctp_faddr_t *, mblk_t *);
1093 extern void       sctp_maxpsz_set(sctp_t *);
1094 extern void       sctp_move_faddr_timers(queue_t *, sctp_t *);

1096 extern sctp_parm_hdr_t *sctp_next_parm(sctp_parm_hdr_t *, ssize_t *);

1098 extern void       sctp_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_regifft_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_ipplen(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, 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 recvg 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

129612 Sat Aug 18 10:37:20 2012

new/usr/src/uts/common/inet/tcp/tcp.c

tcp: conn_mlp_type will be set later in tcp_init_values

_____unchanged portion omitted_____

```
2594 conn_t *
2595 tcp_create_common(cred_t *credp, boolean_t isv6, boolean_t issocket,
2596     int *errorp)
2597 {
2598     tcp_t      *tcp = NULL;
2599     conn_t      *connp;
2600     zoneid_t    zoneid;
2601     tcp_stack_t *tcps;
2602     queue_t     *sqp;
2603
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;
2621
2622         if ((err = secpolicy_basic_net_access(credp)) != 0) {
2623             *errorp = err;
2624             return (NULL);
2625         }
2626
2627         ns = netstack_find_by_cred(credp);
2628         ASSERT(ns != NULL);
2629         tcps = ns->netstack_tcp;
2630         ASSERT(tcps != NULL);
2631
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     }
2642
2643     sqp = IP_QUEUE_GET((uint_t)gethrtime());
2644     connp = (conn_t *)tcp_get_conn(sqp, tcps);
2645     /*
2646      * Both tcp_get_conn and netstack_find_by_cred incremented refcnt,
2647      * so we drop it by one.
2648      */
2649     netstack_rele(tcps->tcps_netstack);
2650     if (connp == NULL) {
2651         *errorp = ENOSR;
2652         return (NULL);
2653     }
```

1

new/usr/src/uts/common/inet/tcp/tcp.c

```
2653     }
2654     ASSERT(connp->conn_ixa->ixa_protocol == connp->conn_proto);
2655
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;
2660
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;
2671
2672     if (!tcps->tcps_dev_flow_ctl)
2673         connp->conn_ixa->ixa_flags |= IXAF_NO_DEV_FLOW_CTL;
2674
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;
2688
2689     crhold(credp);
2690     connp->conn_cred = credp;
2691     connp->conn_cpid = curproc->p_pid;
2692     connp->conn_open_time = ddi_get_lbolt64();
2693
2694     /* Cache things in the ixa without any rehold */
2695     ASSERT(!(connp->conn_ixa->ixa_free_flags & IXA_FREE_CRED));
2696     connp->conn_ixa->ixa_cred = credp;
2697     connp->conn_ixa->ixa_cpid = connp->conn_cpid;
2698
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;
2703     ASSERT(connp->conn_netstack == tcps->tcps_netstack);
2704     ASSERT(tcp->tcp_tcps == tcps);
2705
2706     /*
2707      * If the caller has the process-wide flag set, then default to MAC
2708      * exempt mode. This allows read-down to unlabeled hosts.
2709      */
2710     if (getpflags(NET_MAC_AWARE, credp) != 0)
2711         connp->conn_mac_mode = CONN_MAC_AWARE;
2712
2713     connp->conn_zone_is_global = (crgetzoneid(credp) == GLOBAL_ZONEID);
2714
2715     if (issocket) {
2716         tcp->tcp_issocket = 1;
2717     }
```

2

```
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;
2724
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

new/usr/src/uts/common/inet/tcp/tcp_opt_data.c

1

```
*****
30240 Sat Aug 18 10:37:20 2012
new/usr/src/uts/common/inet/tcp/tcp_opt_data.c
tcp: maybe related to 721fffe3
*****

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_KEEPAALIVE, SOL_SOCKET, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
```

new/usr/src/uts/common/inet/tcp/tcp_opt_data.c

2

```
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_KEEPAALIVE_THRESHOLD, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0,
122   sizeof (int), 0 },

124 { TCP_KEEPIDLE, 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_KEEPINTVL, IPPROTO_TCP, OA_RW, OA_RW, OP_NP, 0, sizeof (int), 0 },
130 { TCP_KEEPA_LIVE_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          /*NOTREACHED*/
300      default:
301          return (B_FALSE);
302      }
303      /*NOTREACHED*/
304  }
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));
416
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));
427
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;
467
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 }
489
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);
508
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;
514
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 (ENOBUFFS);
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 (ENOBUFFS);
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_KEEPIPLE is in seconds but TCP_KEEPAIVE_THRESHOLD
727     * is in milliseconds. TCP_KEEPIPLE is introduced for
728     * compatibility with other Unix flavors.
729     * We can fall through TCP_KEEPAIVE_THRESHOLD logic after
730     * converting the input to milliseconds.
731     */
732     case TCP_KEEPIPLE:
733         *il *= 1000;
734         /* FALLTHRU */
735
736     case TCP_KEEPAIVE_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_KEEPIPLE 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         } else if (tcp->tcp_ka_rinterval == 0) {
776             if ((tcp->tcp_ka_abort_thres / *il) <
777                 tcp->tcp_rto_min ||
778                 (tcp->tcp_ka_abort_thres / *il) >
779                 tcp->tcp_rto_max)
780                 return (EINVAL);
781
782             tcp->tcp_ka_rinterval =
783                 tcp->tcp_ka_abort_thres / *il;
784         } else {
785             if ((*il * tcp->tcp_ka_rinterval) <
786                 tcps->tcps_keepalive_abort_interval_low ||
787                 (*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 }
1034 reterr = conn_opt_set(&coas, level, name, inlen, invalp,
1035                     checkonly, cr);
1036 if (reterr != 0) {
1037     *outlenp = 0;
1038     return (reterr);
1039 }
1040
1041 /*
1042  * Common case of OK return with outval same as inval
1043  */
1044 if (invalp != outvalp) {
1045     /* don't trust bcopy for identical src/dst */
1046     (void) bcopy(invalp, outvalp, inlen);
1047 }
1048 *outlenp = inlen;
1049
1050 if (coas.coa_changed & COA_HEADER_CHANGED) {
1051     /* If we are connected we rebuilt the headers */
1052     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_OOBINLINE_CHANGED) {
1088         if (IPCL_IS_NONSTR(connp))
1089             proto_set_rx_oob_opt(connp, onoff);
1090     }
1091     return (0);
1092 }
```

```

*****
31920 Sat Aug 18 10:37:21 2012
new/usr/src/uts/common/inet/tcp/tcp_socket.c
tcp: maybe related to 721fffe3
*****

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/squeue_impl.h>
35 #include <sys/squeue.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 nmsghdr *,
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

276 static int
277 tcp_connect(sock_lower_handle_t proto_handle, const struct sockaddr *sa,
278             socklen_t len, sock_connid_t *id, cred_t *cr)
279 {
280     conn_t      *connp = (conn_t *)proto_handle;
281     int         error;

283     ASSERT(connp->conn_upper_handle != NULL);

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

288     error = proto_verify_ip_addr(connp->conn_family, sa, len);
289     if (error != 0) {
290         return (error);
291     }

293     error = squeue_synch_enter(connp, NULL);
294     if (error != 0) {
295         /* failed to enter */
296         return (ENOSR);
297     }

299     /*
300      * TCP supports quick connect, so no need to do an implicit bind
301      */
302     error = tcp_do_connect(connp, sa, len, cr, curproc->p_pid);
303     if (error == 0) {
304         *id = connp->conn_tcp->tcp_connid;
305     } else if (error < 0) {
306         if (error == -TOUTSTATE) {
307             switch (connp->conn_tcp->tcp_state) {

```

```

308             case TCPS_SYN_SENT:
309                 error = EALREADY;
310                 break;
311             case TCPS_ESTABLISHED:
312                 error = EISCONN;
313                 break;
314             case TCPS_LISTEN:
315                 error = EOPNOTSUPP;
316                 break;
317             default:
318                 error = EINVAL;
319                 break;
320         } else {
321             error = proto_tltosyserr(-error);
322         }
323     }
324 }
325
326 if (connp->conn_tcp->tcp_loopback) {
327     struct sock_proto_props sopp;
328
329     sopp.sopp_flags = SOCKOPT_LOOPBACK;
330     sopp.sopp_loopback = B_TRUE;
331
332     (*connp->conn_upcalls->su_set_proto_props)(
333         connp->conn_upper_handle, &sopp);
334 }
335 done:
336     squeue_synch_exit(connp);
337
338     return ((error == 0) ? EINPROGRESS : error);
339 }
340
341 unchanged_portion_omitted

```

```

752 /* ARGSUSED */
753 sock_lower_handle_t
754 tcp_create(int family, int type, int proto, sock_downcalls_t **sock_downcalls,
755            uint_t *smodep, int *errorp, int flags, cred_t *credp)
756 {
757     conn_t      *connp;
758     boolean_t    isv6 = family == AF_INET6;
759
760 #endif /* ! codereview */
761     if (type != SOCK_STREAM || (family != AF_INET && family != AF_INET6) ||
762         (proto != 0 && proto != IPPROTO_TCP)) {
763         *errorp = EPROTONOSUPPORT;
764         return (NULL);
765     }
766
767     connp = tcp_create_common(credp, isv6, B_TRUE, errorp);
768     if (connp == NULL) {
769         return (NULL);
770     }
771
772     /*
773      * Put the ref for TCP. Ref for IP was already put
774      * by ipcl_conn_create. Also make the conn_t globally
775      * visible to walkers.
776      * by ipcl_conn_create. Also Make the conn_t globally
777      * visible to walkers
778      */
779     mutex_enter(&connp->conn_lock);
780     CONN_INC_REF_LOCKED(connp);
781     ASSERT(connp->conn_ref == 2);
782     connp->conn_state_flags &= ~CONN_INCIPIENT;

```

```

782     connp->conn_flags |= IPCL_NONSTR;
783     mutex_exit(&connp->conn_lock);
784
785     ASSERT(errorp != NULL);
786     *errorp = 0;
787     *sock_downcalls = &sock_tcp_downcalls;
788     *smodep = SM_CONNREQUIRED | SM_EXDATA | SM_ACCEPTSUPP |
789             SM_SENDFILESUPP;
790
791     return ((sock_lower_handle_t)connp);
792 }
793
794 unchanged_portion_omitted

```

```

*****
28387 Sat Aug 18 10:37:21 2012
new/usr/src/uts/common/inet/tcp_impl.h
tcp: maybe related to 721ffe3
*****
_____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 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 \
444     tcps_propinfo_tbl[20].prop_max_uval
445 #define tcps_rexmit_interval_initial \
446     tcps_propinfo_tbl[20].prop_cur_uval
447 #define tcps_rexmit_interval_initial_low \
448     tcps_propinfo_tbl[20].prop_min_uval
449 #define tcps_rexmit_interval_max_high tcps_propinfo_tbl[21].prop_max_uval
450 #define tcps_rexmit_interval_max tcps_propinfo_tbl[21].prop_cur_uval
451 #define tcps_rexmit_interval_max_low tcps_propinfo_tbl[21].prop_min_uval
452 #define tcps_rexmit_interval_min_high tcps_propinfo_tbl[22].prop_max_uval
453 #define tcps_rexmit_interval_min tcps_propinfo_tbl[22].prop_cur_uval
454 #define tcps_rexmit_interval_min_low tcps_propinfo_tbl[22].prop_min_uval
455 #define tcps_deferred_ack_interval tcps_propinfo_tbl[23].prop_cur_uval
456 #define tcps_snd_lowat_fraction tcps_propinfo_tbl[24].prop_cur_uval
457 #define tcps_dupack_fast_retransmit tcps_propinfo_tbl[25].prop_cur_uval
458 #define tcps_ignore_path_mtu tcps_propinfo_tbl[26].prop_cur_bval
459 #define tcps_smallest_anon_port tcps_propinfo_tbl[27].prop_cur_uval
460 #define tcps_largest_anon_port tcps_propinfo_tbl[28].prop_cur_uval
461 #define tcps_xmit_hiwat tcps_propinfo_tbl[29].prop_cur_uval
462 #define tcps_xmit_lowat tcps_propinfo_tbl[30].prop_cur_uval
463 #define tcps_rcv_hiwat tcps_propinfo_tbl[31].prop_cur_uval
464 #define tcps_rcv_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         tcps_propinfo_tbl[33].prop_cur_uval
467 #define tcps_fin_wait_2_flush_interval_low \
468         tcps_propinfo_tbl[33].prop_min_uval
469 #define tcps_max_buf         tcps_propinfo_tbl[34].prop_cur_uval
470 #define tcps_strong_iss      tcps_propinfo_tbl[35].prop_cur_uval
471 #define tcps_rtt_updates     tcps_propinfo_tbl[36].prop_cur_uval
472 #define tcps_wscales_always  tcps_propinfo_tbl[37].prop_cur_bval
473 #define tcps_tstamp_always   tcps_propinfo_tbl[38].prop_cur_bval
474 #define tcps_tstamp_if_wscales tcps_propinfo_tbl[39].prop_cur_bval
475 #define tcps_rexmit_interval_extra tcps_propinfo_tbl[40].prop_cur_uval
476 #define tcps_deferred_acks_max tcps_propinfo_tbl[41].prop_cur_uval
477 #define tcps_slow_start_after_idle tcps_propinfo_tbl[42].prop_cur_uval
478 #define tcps_slow_start_initial tcps_propinfo_tbl[43].prop_cur_uval
479 #define tcps_sack_permitted   tcps_propinfo_tbl[44].prop_cur_uval
480 #define tcps_ipv6_hoplimit    tcps_propinfo_tbl[45].prop_cur_uval
481 #define tcps_mss_def_ipv6     tcps_propinfo_tbl[46].prop_cur_uval
482 #define tcps_mss_max_ipv6     tcps_propinfo_tbl[47].prop_cur_uval
483 #define tcps_rev_src_routes    tcps_propinfo_tbl[48].prop_cur_bval
484 #define tcps_local_dack_interval tcps_propinfo_tbl[49].prop_cur_uval
485 #define tcps_local_dacks_max   tcps_propinfo_tbl[50].prop_cur_uval
486 #define tcps_ecn_permitted     tcps_propinfo_tbl[51].prop_cur_uval
487 #define tcps_rst_sent_rate_enabled tcps_propinfo_tbl[52].prop_cur_bval
488 #define tcps_rst_sent_rate     tcps_propinfo_tbl[53].prop_cur_uval
489 #define tcps_push_timer_interval tcps_propinfo_tbl[54].prop_cur_uval
490 #define tcps_use_smss_as_mss_opt tcps_propinfo_tbl[55].prop_cur_bval
491 #define tcps_keeppalive_abort_interval_high \
492         tcps_propinfo_tbl[56].prop_max_uval
493 #define tcps_keeppalive_abort_interval \
494         tcps_propinfo_tbl[56].prop_cur_uval
495 #define tcps_keeppalive_abort_interval_low \
496         tcps_propinfo_tbl[56].prop_min_uval
497 #define tcps_wroff_xtra       tcps_propinfo_tbl[57].prop_cur_uval
498 #define tcps_dev_flow_ctl     tcps_propinfo_tbl[58].prop_cur_bval
499 #define tcps_reass_timeout    tcps_propinfo_tbl[59].prop_cur_uval
500 #define tcps_iss_incr         tcps_propinfo_tbl[65].prop_cur_uval

502 extern struct qinit tcp_rinitv4, tcp_rinitv6;
503 extern boolean_t do_tcp_fusion;

505 /*
506  * Object to represent database of options to search passed to
507  * {sock, tpi}optcom_req() interface routine to take care of option
508  * management and associated methods.
509  */
510 extern optdb_obj_t      tcp_opt_obj;
511 extern uint_t           tcp_max_optsize;

513 extern int tcp_squeue_flag;

515 extern uint_t tcp_free_list_max_cnt;

517 /*
518  * Functions in tcp.c.
519  */
520 extern void tcp_acceptor_hash_insert(t_uscalar_t, tcp_t *);
521 extern tcp_t *tcp_acceptor_hash_lookup(t_uscalar_t, tcp_stack_t *);
522 extern void tcp_acceptor_hash_remove(tcp_t *);
523 extern mblk_t *tcp_ack_mp(tcp_t *);
524 extern int tcp_build_hdrs(tcp_t *);
525 extern void tcp_cleanup(tcp_t *);
526 extern int tcp_clean_death(tcp_t *, int);
527 extern void tcp_clean_death_wrapper(void *, mblk_t *, void *,
528         ip_rcv_attr_t *);
529 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 *);

566 /*
567  * Bind related functions in tcp_bind.c
568  */
569 extern int tcp_bind_check(conn_t *, struct sockaddr *, socklen_t,
570         cred_t *, boolean_t);
571 extern void tcp_bind_hash_insert(tf_t *, tcp_t *, int);
572 extern void tcp_bind_hash_remove(tcp_t *);
573 extern in_port_t tcp_bindi(tcp_t *, in_port_t, const in6_addr_t *,
574         int, boolean_t, boolean_t, boolean_t);
575 extern in_port_t tcp_update_next_port(in_port_t, const tcp_t *,
576         boolean_t);

578 /*
579  * Fusion related functions in tcp_fusion.c.
580  */
581 extern void tcp_fuse(tcp_t *, uchar_t *, tpha_t *);
582 extern void tcp_unfuse(tcp_t *);
583 extern boolean_t tcp_fuse_output(tcp_t *, mblk_t *, uint32_t);
584 extern void tcp_fuse_output_urg(tcp_t *, mblk_t *);
585 extern boolean_t tcp_fuse_rcv_drain(queue_t *, tcp_t *, mblk_t **);
586 extern size_t tcp_fuse_set_rcv_hiwat(tcp_t *, size_t);
587 extern int tcp_fuse_maxpsz(tcp_t *);
588 extern void tcp_fuse_backenable(tcp_t *);
589 extern void tcp_iss_key_init(uint8_t *, int, tcp_stack_t *);

591 /*

```

```

592 * Output related functions in tcp_output.c.
593 */
594 extern void tcp_close_output(void *, mblk_t *, void *, ip_recv_attr_t *);
595 extern void tcp_output(void *, mblk_t *, void *, ip_recv_attr_t *);
596 extern void tcp_output_urgent(void *, mblk_t *, void *, ip_recv_attr_t *);
597 extern void tcp_rexmit_after_error(tcp_t *);
598 extern void tcp_sack_rexmit(tcp_t *, uint_t *);
599 extern void tcp_send_data(tcp_t *, mblk_t *);
600 extern void tcp_send_synack(void *, mblk_t *, void *, ip_recv_attr_t *);
601 extern void tcp_shutdown_output(void *, mblk_t *, void *, ip_recv_attr_t *);
602 extern void tcp_ss_rexmit(tcp_t *);
603 extern void tcp_update_xmit_tail(tcp_t *, uint32_t);
604 extern void tcp_wput(queue_t *, mblk_t *);
605 extern void tcp_wput_data(tcp_t *, mblk_t *, boolean_t);
606 extern void tcp_wput_sock(queue_t *, mblk_t *);
607 extern void tcp_wput_fallback(queue_t *, mblk_t *);
608 extern void tcp_xmit_ctl(char *, tcp_t *, uint32_t, uint32_t, int);
609 extern void tcp_xmit_listeners_reset(mblk_t *, ip_recv_attr_t *,
610 ip_stack_t *, conn_t *);
611 extern mblk_t *tcp_xmit_mp(tcp_t *, mblk_t *, int32_t, int32_t *,
612 mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);

614 /*
615 * Input related functions in tcp_input.c.
616 */
617 extern void tcp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
618 extern void tcp_input_data(void *, mblk_t *, void *, ip_recv_attr_t *);
619 extern void tcp_input_listener_unbound(void *, mblk_t *, void *,
620 ip_recv_attr_t *);
621 extern boolean_t tcp_paws_check(tcp_t *, tcp_h_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 *, icmp_h_t *, icmp6_t *,
625 ip_recv_attr_t *);

627 /*
628 * Kernel socket related functions in tcp_socket.c.
629 */
630 extern int tcp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
631 so_proto_quiesced_cb_t, sock_quiesce_arg_t *);
632 extern boolean_t tcp_newconn_notify(tcp_t *, ip_recv_attr_t *);

634 /*
635 * Timer related functions in tcp_timers.c.
636 */
637 extern void tcp_ack_timer(void *);
638 extern void tcp_close_linger_timeout(void *);
639 extern void tcp_keeppalive_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_recv_attr_t *);
656 extern void tcp_err_ack(tcp_t *, mblk_t *, int, int);
657 extern void tcp_err_ack_prim(tcp_t *, mblk_t *, int, int, int);

```

```

658 extern void tcp_info_req(tcp_t *, mblk_t *);
659 extern void tcp_send_conn_ind(void *, mblk_t *, void *);
660 extern void tcp_send_pending(void *, mblk_t *, void *, ip_recv_attr_t *);
661 extern void tcp_tpi_accept(queue_t *, mblk_t *);
662 extern void tcp_tpi_bind(tcp_t *, mblk_t *);
663 extern int tcp_tpi_close(queue_t *, int);
664 extern int tcp_tpi_close_accept(queue_t *);
665 extern void tcp_tpi_connect(tcp_t *, mblk_t *);
666 extern int tcp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
667 extern int tcp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
668 uint_t *, uchar_t *, void *, cred_t *);
669 extern void tcp_tpi_unbind(tcp_t *, mblk_t *);
670 extern void tcp_tli_accept(tcp_t *, mblk_t *);
671 extern void tcp_use_pure_tpi(tcp_t *);
672 extern void tcp_do_capability_ack(tcp_t *, struct T_capability_ack *,
673 t_uscalar_t);

675 /*
676 * TCP option processing related functions in tcp_opt_data.c
677 */
678 extern int tcp_opt_default(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
679 extern int tcp_opt_get(conn_t *, int, int, uchar_t *);
680 extern int tcp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
681 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, tcp_h_t *, ip_recv_attr_t *);

691 /*
692 * Misc functions in tcp_misc.c.
693 */
694 extern uint32_t tcp_find_listener_conf(tcp_stack_t *, in_port_t);
695 extern void tcp_ioctl_abort_conn(queue_t *, mblk_t *);
696 extern void tcp_listener_conf_cleanup(tcp_stack_t *);
697 extern void tcp_stack_cpu_add(tcp_stack_t *, processorid_t);

699 #endif /* _KERNEL */

701 #ifdef __cplusplus
702 }

```

unchanged portion omitted

new/usr/src/uts/common/inet/tcp_stats.h

1

```
*****
7689 Sat Aug 18 10:37:21 2012
new/usr/src/uts/common/inet/tcp_stats.h
tcp: spelling
*****

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  * 1) Add a field in the tcp_stat structure describing your counter.
60  * 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_wsrvt_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_unfusabed;
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;
```

new/usr/src/uts/common/inet/tcp_stats.h

3

```
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
```



```

*****
11287 Sat Aug 18 10:37:21 2012
new/usr/src/uts/common/inet/tunables.c
dccp: starting module template
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
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_stack.h>
38 #include <inet/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 }

```

new/usr/src/uts/common/inet/tunables.h

1

```
*****
6181 Sat Aug 18 10:37:22 2012
new/usr/src/uts/common/inet/tunables.h
dccp: starting module template
*****
_____unchanged_portion_omitted_____

59 #define MOD_PROP_VERSION      1

61 /* permission flags for properties */
62 #define MOD_PROP_PERM_READ     0x1
63 #define MOD_PROP_PERM_WRITE    0x2
64 #define MOD_PROP_PERM_RW      (MOD_PROP_PERM_READ|MOD_PROP_PERM_WRITE)

66 /* mpr_flags values */
67 #define MOD_PROP_ACTIVE        0x01 /* current value of the property */
68 #define MOD_PROP_DEFAULT      0x02 /* default value of the property */
69 #define MOD_PROP_POSSIBLE     0x04 /* possible values for the property */
70 #define MOD_PROP_PERM         0x08 /* read/write permission for property */
71 #define MOD_PROP_APPEND       0x10 /* append to multi-valued property */
72 #define MOD_PROP_REMOVE       0x20 /* remove from multi-valued property */

74 /* mpr_proto values */
75 #define MOD_PROTO_NONE        0x00
76 #define MOD_PROTO_IPV4        0x01 /* property is applicable to IPV4 */
77 #define MOD_PROTO_IPV6        0x02 /* property is applicable to IPV6 */
78 #define MOD_PROTO_RAWIP       0x04 /* property is applicable to ICMP */
79 #define MOD_PROTO_TCP         0x08 /* property is applicable to TCP */
80 #define MOD_PROTO_UDP         0x10 /* property is applicable to UDP */
81 #define MOD_PROTO_SCTP        0x20 /* property is applicable to SCTP */
82 #define MOD_PROTO_DCCP        0x40 /* property is applicable to DCCP */
83 #endif /* ! codereview */

85 /* property is applicable to both IPV[4|6] */
86 #define MOD_PROTO_IP          (MOD_PROTO_IPV4|MOD_PROTO_IPV6)

88 #ifdef _KERNEL

90 typedef struct mod_prop_info_s mod_prop_info_t;

92 /* set/get property callback functions */
93 typedef int      mod_prop_setf_t(void *, cred_t *, mod_prop_info_t *,
94     const char *, const void *, uint_t);
95 typedef int      mod_prop_getf_t(void *, mod_prop_info_t *, const char *,
96     void *val, uint_t, uint_t);

98 typedef struct mod_propval_uint32_s {
99     uint32_t      mod_propval_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;
```

new/usr/src/uts/common/inet/tunables.h

2

```
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/Makefile

1

1543 Sat Aug 18 10:37:22 2012

new/usr/src/uts/common/netinet/Makefile

dccp: snoop, build system fixes

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 #ident      "%Z%M% %I%      %E% SMI"
23 #
24 # Copyright 2007 Sun Microsystems, Inc. All rights reserved.
25 # Use is subject to license terms.
26 #
27 #
28 # uts/common/netinet/Makefile
29 #
30 # include global definitions
31 # include ../../../../Makefile.master
```

```
29 HDRS=      arp.h dccp.h dhcp.h dhcp6.h icmp6.h icmp_var.h if_ether.h igmp.h \
30            igmp_var.h in.h inetutil.h in_pcb.h in_sysm.h in_var.h ip.h ip6.h \
31            ip_icmp.h ip_mroute.h ip_var.h pim.h sctp.h tcp.h tcp_debug.h \
32            tcp_fsm.h tcp_seq.h tcp_timer.h tcp_var.h tcpip.h udp.h udp_var.h
33 HDRS=      arp.h dhcp.h dhcp6.h icmp6.h icmp_var.h if_ether.h igmp.h igmp_var.h \
34            in.h inetutil.h in_pcb.h in_sysm.h in_var.h ip.h ip6.h ip_icmp.h \
35            ip_mroute.h ip_var.h pim.h sctp.h tcp.h tcp_debug.h tcp_fsm.h \
36            tcp_seq.h tcp_timer.h tcp_var.h tcpip.h udp.h udp_var.h
```

34 ROOTDIRS= \$(ROOT)/usr/include/netinet

36 Roothdrs= \$(HDRS:%=\$(ROOT)/usr/include/netinet/%)

38 CHECKHDRS= \$(HDRS:%.h=%.check)

40 \$(ROOTDIRS)/%: %
41 \$(INS.file)

43 .KEEP_STATE:

45 .PARALLEL: \$(CHECKHDRS)

47 install_h: \$(ROOTDIRS) \$(Roothdrs)

49 \$(ROOTDIRS):
50 \$(INS.dir)

52 check: \$(CHECKHDRS)

new/usr/src/uts/common/netinet/dccp.h

1

1948 Sat Aug 18 10:37:22 2012
new/usr/src/uts/common/netinet/dccp.h
dccp: starting module template

```
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 Hoeppner. 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 header
25 */
26 struct dccphdr {
27     uint16_t      dh_sport;
28     uint16_t      dh_dport;
29     uint8_t       dh_offset;
30 #ifdef _BIT_FIELDS_LTOH
31     uint8_t       dh_cscov:4,
32                 dh_ccval:4;
33 #else
34     uint8_t       dh_ccval:4,
35                 dh_cscov:4;
36 #endif
37     uint16_t      dh_sum;
38 #ifdef _BIT_FIELDS_LTOH
39     uint8_t       dh_x:1,
40                 dh_type:4,
41                 dh_reserved:3;
42 #else
43     uint8_t       dh_reserved:3,
44                 dh_type:4,
45                 dh_x:1;
46 #endif
47     uint8_t       dh_res_seq;
48     uint16_t      dh_seq;
49 };
```

```
51 #define DCCPOPT_PADDING          0
52 #define DCCPOPT_MANDATORY        1
53 #define DCCPOPT_SLOW_RECEIVER    2
54 #define DCCPOPT_CHANGE_L        32
55 #define DCCPOPT_CONFIRM_L       33
56 #define DCCPOPT_CHANGE_R        34
57 #define DCCPOPT_CONFIRM_R       35
58 #define DCCPOPT_INIT_COOKIE     36
59 #define DCCPOPT_NDP_COUNT       37
60 #define DCCPOPT_ACK_VECTOR_1     38
61 #define DCCPOPT_ACK_VECTOR_2     39
```

new/usr/src/uts/common/netinet/dccp.h

2

```
62 #define DCCPOPT_DATA_DROPPED     40
63 #define DCCPOPT_TIMESTAMP        41
64 #define DCCPOPT_TIMESTAMP_ECHO   42
65 #define DCCPOPT_ELAPSED_TIME     43
66 #define DCCPOPT_DATA_CHECKSUM    44
```

```
68 /*
69  * Options for use with getsockopt and setsockopt.
70 */
```

```
72 #define DCCP_KEEPAIVE             0x08
73 #define DCCP_NOTIFY_THRESHOLD     0x10
74 #define DCCP_ABORT_THRESHOLD     0x11
75 #define DCCP_CONN_NOTIFY_THRESHOLD 0x12
76 #define DCCP_CONN_ABORT_THRESHOLD 0x13
```

```
78 #define DCCP_KEEPAIVE_THRESHOLD  0x16
79 #define DCCP_KEEPAIVE_ABORT_THRESHOLD 0x17
```

```
81 #define DCCP_KEEPIPLE             0x22
82 #define DCCP_KEEPCNT              0x23
83 #define DCCP_KEEPIVTL             0x24
```

```
85 #ifdef __cplusplus
86 }
87 #endif
```

```
89 #endif /* _NETINET_DCCP_H */
90 #endif /* ! codereview */
```


new/usr/src/uts/common/netinet/in.h

1

```
*****
44002 Sat Aug 18 10:37:23 2012
new/usr/src/uts/common/netinet/in.h
dccp: starting module template
*****
__unchanged_portion_omitted__
128 #define s6_addr      _S6_un._S6_u8

130 #ifdef _KERNEL
131 #define s6_addr8      _S6_un._S6_u8
132 #define s6_addr32    _S6_un._S6_u32
133 #endif

135 typedef struct in6_addr in6_addr_t;

137 #endif /* !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__) */

139 #ifndef _SA_FAMILY_T
140 #define _SA_FAMILY_T
141 typedef uint16_t      sa_family_t;
142 #endif

144 /*
145  * Protocols
146  *
147  * Some of these constant names are copied for the DTrace IP provider in
148  * usr/src/lib/libdtrace/common/{ip.d.in, ip.sed.in}, which should be kept
149  * in sync.
150  */
151 #define IPPROTO_IP      0          /* dummy for IP */
152 #define IPPROTO_HOPOPTS 0          /* Hop by hop header for IPv6 */
153 #define IPPROTO_ICMP    1          /* control message protocol */
154 #define IPPROTO_IGMP    2          /* group control protocol */
155 #define IPPROTO_GGP     3          /* gateway^2 (deprecated) */
156 #define IPPROTO_ENCAP   4          /* IP in IP encapsulation */
157 #define IPPROTO_TCP     6          /* tcp */
158 #define IPPROTO_EGP     8          /* exterior gateway protocol */
159 #define IPPROTO_PUP     12         /* pup */
160 #define IPPROTO_UDP     17         /* user datagram protocol */
161 #define IPPROTO_IDP     22         /* xns idp */
162 #define IPPROTO_DCCP    33         /* DCCP */
163 #endif /* ! codereview */
164 #define IPPROTO_IPV6    41         /* IPv6 encapsulated in IP */
165 #define IPPROTO_ROUTING 43         /* Routing header for IPv6 */
166 #define IPPROTO_FRAGMENT 44        /* Fragment header for IPv6 */
167 #define IPPROTO_RSVP    46         /* rsvp */
168 #define IPPROTO_ESP     50         /* IPsec Encap. Sec. Payload */
169 #define IPPROTO_AH      51         /* IPsec Authentication Hdr. */
170 #define IPPROTO_ICMPV6  58         /* ICMP for IPv6 */
171 #define IPPROTO_NONE    59         /* No next header for IPv6 */
172 #define IPPROTO_DSTOPTS 60         /* Destination options */
173 #define IPPROTO_HELLO   63         /* "hello" routing protocol */
174 #define IPPROTO_ND      77         /* UNOFFICIAL net disk proto */
175 #define IPPROTO_EON     80         /* ISO clnp */
176 #define IPPROTO OSPF    89         /* OSPF */
177 #define IPPROTO_PIM     103        /* PIM routing protocol */
178 #define IPPROTO SCTP    132        /* Stream Control */
179                                     /* Transmission Protocol */

181 #define IPPROTO_RAW     255        /* raw IP packet */
182 #define IPPROTO_MAX     256

184 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
185 #define PROTO_SDP      257        /* Sockets Direct Protocol */
186 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
```

new/usr/src/uts/common/netinet/in.h

2

```
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 IPPROTO_ECHO      7
195 #define IPPROTO_DISCARD  9
196 #define IPPROTO_SYSTAT  11
197 #define IPPROTO_DAYTIME  13
198 #define IPPROTO_NETSTAT  15
199 #define IPPROTO_CHARGEN  19
200 #define IPPROTO_FTP      21
201 #define IPPROTO_TELNET   23
202 #define IPPROTO_SMTTP    25
203 #define IPPROTO_TIMESERVER 37
204 #define IPPROTO_NAMESERVER 42
205 #define IPPROTO_WHOIS    43
206 #define IPPROTO_DOMAIN   53
207 #define IPPROTO_MDNS     5353
208 #define IPPROTO_MTP      57

210 /*
211  * Port/socket numbers: host specific functions
212  */
213 #define IPPROTO_BOOTPS    67
214 #define IPPROTO_BOOTPC    68
215 #define IPPROTO_TFTP      69
216 #define IPPROTO_RJE       77
217 #define IPPROTO_FINGER    79
218 #define IPPROTO_HTTP      80
219 #define IPPROTO_HTTP_ALT  8080
220 #define IPPROTO_TTYLINK   87
221 #define IPPROTO_SUPDUP    95
222 #define IPPROTO_NTP      123
223 #define IPPROTO_NETBIOS_NS 137
224 #define IPPROTO_NETBIOS_DGM 138
225 #define IPPROTO_NETBIOS_SSN 139
226 #define IPPROTO_LDAP     389
227 #define IPPROTO_SLP      427
228 #define IPPROTO_MIP      434
229 #define IPPROTO_SMB      445          /* a.k.a. microsoft-ds */

231 /*
232  * Internet Key Exchange (IKE) ports
233  */
234 #define IPPROTO_IKE       500
235 #define IPPROTO_IKE_NATT  4500

237 /*
238  * UNIX TCP sockets
239  */
240 #define IPPROTO_EXECSERVER 512
241 #define IPPROTO_LOGIN SERVER 513
242 #define IPPROTO_CMDSERVER 514
243 #define IPPROTO_PRINTER  515
244 #define IPPROTO_EFS SERVER 520

246 /*
247  * UNIX UDP sockets
248  */
249 #define IPPROTO_BIFFUDP    512
250 #define IPPROTO_WHOSE SERVER 513
251 #define IPPROTO_SYSLOG    514
252 #define IPPROTO_TALK      517
253 #define IPPROTO_ROUTE SERVER 520
```

```

254 #define IPPORT_RIPNG          521

256 /*
257  * DHCPv6 UDP ports
258  */
259 #define IPPORT_DHCPV6C          546
260 #define IPPORT_DHCPV6S          547

262 #define IPPORT_SOCKS            1080

264 /*
265  * Ports < IPPORT_RESERVED are reserved for
266  * privileged processes (e.g. root).
267  * Ports > IPPORT_USERRESERVED are reserved
268  * for servers, not necessarily privileged.
269  */
270 #define IPPORT_RESERVED        1024
271 #define IPPORT_USERRESERVED    5000

273 /*
274  * Link numbers
275  */
276 #define IMPLINK_IP              155
277 #define IMPLINK_LOWEXPER        156
278 #define IMPLINK_HIGHEXPER      158

280 /*
281  * IPv4 Internet address
282  * This definition contains obsolete fields for compatibility
283  * with SunOS 3.x and 4.2bsd. The presence of subnets renders
284  * divisions into fixed fields misleading at best. New code
285  * should use only the s_addr field.
286  */

288 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
289 #define _S_un_b _S_un_b
290 #define _S_un_w _S_un_w
291 #define _S_addr _S_addr
292 #define _S_un _S_un
293 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */

295 struct in_addr {
296     union {
297         struct { uint8_t s_b1, s_b2, s_b3, s_b4; } _S_un_b;
298         struct { uint16_t s_w1, s_w2; } _S_un_w;
299         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 };

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

329 #define IN_CLASSB(i)            (((i) & 0xc0000000U) == 0x80000000U)
330 #define IN_CLASSB_NET          0xffff0000U
331 #define IN_CLASSB_NSHIFT       16
332 #define IN_CLASSB_HOST         0x0000ffffU
333 #define IN_CLASSB_MAX          65536

335 #define IN_CLASSC(i)            (((i) & 0xe0000000U) == 0xc0000000U)
336 #define IN_CLASSC_NET          0xffff0000U
337 #define IN_CLASSC_NSHIFT       8
338 #define IN_CLASSC_HOST         0x000000ffU

340 #define IN_CLASSD(i)            (((i) & 0xf0000000U) == 0xe0000000U)
341 #define IN_CLASSD_NET          0xf0000000U /* These aren't really */
342 #define IN_CLASSD_NSHIFT       28 /* net and host fields, but */
343 #define IN_CLASSD_HOST         0x0ffffffU /* routing needn't know */

345 #define IN_CLASSE(i)            (((i) & 0xf0000000U) == 0xf0000000U)
346 #define IN_CLASSE_NET          0xffffffffU

348 #define IN_MULTICAST(i)         IN_CLASSD(i)

350 /*
351  * We have removed CLASS E checks from the kernel
352  * But we preserve these defines for userland in order
353  * to avoid compile breakage of some 3rd party piece of software
354  */
355 #ifndef _KERNEL
356 #define IN_EXPERIMENTAL(i)      (((i) & 0xe0000000U) == 0xe0000000U)
357 #define IN_BADCLASS(i)         (((i) & 0xf0000000U) == 0xf0000000U)
358 #endif

360 #define INADDR_ANY              0x00000000U
361 #define INADDR_LOOPBACK        0x7f000001U
362 #define INADDR_BROADCAST       0xffffffffU /* must be masked */
363 #define INADDR_NONE            0xffffffffU

365 #define INADDR_UNSPEC_GROUP     0xe0000000U /* 224.0.0.0 */
366 #define INADDR_ALLHOSTS_GROUP  0xe0000001U /* 224.0.0.1 */
367 #define INADDR_ALLRTRS_GROUP   0xe0000002U /* 224.0.0.2 */
368 #define INADDR_ALLRPTS_GROUP   0xe0000016U /* 224.0.0.22, IGMPv3 */
369 #define INADDR_MAX_LOCAL_GROUP 0xe00000ffU /* 224.0.0.255 */

371 /* Scoped IPv4 prefixes (in host byte-order) */
372 #define IN_AUTOCONF_NET         0xa9fe0000U /* 169.254/16 */
373 #define IN_AUTOCONF_MASK       0xffff0000U
374 #define IN_PRIVATE8_NET        0xa0a00000U /* 10/8 */
375 #define IN_PRIVATE8_MASK       0xff000000U
376 #define IN_PRIVATE12_NET       0xac100000U /* 172.16/12 */
377 #define IN_PRIVATE12_MASK      0xff000000U
378 #define IN_PRIVATE16_NET       0xc0a80000U /* 192.168/16 */
379 #define IN_PRIVATE16_MASK      0xffff0000U

381 /* RFC 3927 IPv4 link local address (i in host byte-order) */
382 #define IN_LINKLOCAL(i)         (((i) & IN_AUTOCONF_MASK) == IN_AUTOCONF_NET)

384 /* Well known 6to4 Relay Router Anycast address defined in RFC 3068 */
385 #if !defined(_XPG4_2) || !defined(__EXTENSIONS__)

```

```

386 #define INADDR_6TO4RRANYCAST 0xc0586301U /* 192.88.99.1 */
387 #endif /* !defined(_XPG4_2) || !defined(__EXTENSIONS__) */

389 #define IN_LOOPBACKNET 127 /* official! */

391 /*
392  * Define a macro to stuff the loopback address into an Internet address
393  */
394 #if !defined(_XPG4_2) || !defined(__EXTENSIONS__)
395 #define IN_SET_LOOPBACK_ADDR(a) \
396 { (a)->sin_addr.s_addr = htonl(INADDR_LOOPBACK); \
397   (a)->sin_family = AF_INET; }
398 #endif /* !defined(_XPG4_2) || !defined(__EXTENSIONS__) */

400 /*
401  * IPv4 Socket address.
402  */
403 struct sockaddr_in {
404     sa_family_t    sin_family;
405     in_port_t      sin_port;
406     struct in_addr  sin_addr;
407 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
408     char           sin_zero[8];
409 #else
410     unsigned char  sin_zero[8];
411 #endif /* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
412 };

414 #if !defined(_XPG4_2) || defined(_XPG6) || defined(__EXTENSIONS__)
415 /*
416  * IPv6 socket address.
417  */
418 struct sockaddr_in6 {
419     sa_family_t    sin6_family;
420     in_port_t      sin6_port;
421     uint32_t       sin6_flowinfo;
422     struct in6_addr sin6_addr;
423     uint32_t       sin6_scope_id; /* Depends on scope of sin6_addr */
424     uint32_t       __sin6_src_id; /* Impl. specific - UDP replies */
425 };

427 /*
428  * Macros for accessing the traffic class and flow label fields from
429  * sin6_flowinfo.
430  * These are designed to be applied to a 32-bit value.
431  */
432 #ifndef _BIG_ENDIAN
433 /* masks */
434 #define IPV6_FLOWINFO_FLOWLABEL 0x000fffffu
435 #define IPV6_FLOWINFO_TCLASS 0x0ff00000u

437 #else /* _BIG_ENDIAN */
438 /* masks */
439 #define IPV6_FLOWINFO_FLOWLABEL 0xffff0f00u
440 #define IPV6_FLOWINFO_TCLASS 0x0000f00fu

442 #endif /* _BIG_ENDIAN */

444 /*
445  * Note: Macros IN6ADDR_ANY_INIT and IN6ADDR_LOOPBACK_INIT are for
446  * use as RHS of Static initializers of "struct in6_addr" (or in6_addr_t)
447  * only. They need to be different for User/Kernel versions because union
448  * component data structure is defined differently (it is identical at
449  * binary representation level).

```

```

452 *
453 * const struct in6_addr IN6ADDR_ANY_INIT;
454 * const struct in6_addr IN6ADDR_LOOPBACK_INIT;
455 */

457 #ifndef _KERNEL
458 #define IN6ADDR_ANY_INIT { 0, 0, 0, 0 }

459 #define IN6ADDR_LOOPBACK_INIT { 0, 0, 0, 0x00000001u }
460 #else /* _BIG_ENDIAN */
461 #define IN6ADDR_LOOPBACK_INIT { 0, 0, 0, 0x01000000u }
462 #endif /* _BIG_ENDIAN */

463 #else
464 #define IN6ADDR_ANY_INIT { 0, 0, 0, 0, \
465                           0, 0, 0, 0, \
466                           0, 0, 0, 0, \
467                           0, 0, 0, 0 }
468 #define IN6ADDR_LOOPBACK_INIT { 0, 0, 0, 0, \
469                                 0, 0, 0, 0, \
470                                 0, 0, 0, 0, \
471                                 0, 0, 0, 0x1u }
472 #endif /* _KERNEL */

473 /*
474  * RFC 2553 specifies the following macros. Their type is defined
475  * as "int" in the RFC but they only have boolean significance
476  * (zero or non-zero). For the purposes of our comment notation,
477  * we assume a hypothetical type "bool" defined as follows to
478  * write the prototypes assumed for macros in our comments better.
479  * typedef int bool;
480  */

481 /*
482  * IN6 macros used to test for special IPv6 addresses
483  * (Mostly from spec)
484  */
485 #define IN6_IS_ADDR_UNSPECIFIED(addr) (const struct in6_addr *)
486 #define IN6_IS_ADDR_LOOPBACK(addr) (const struct in6_addr *)
487 #define IN6_IS_ADDR_MULTICAST(addr) (const struct in6_addr *)
488 #define IN6_IS_ADDR_LINKLOCAL(addr) (const struct in6_addr *)
489 #define IN6_IS_ADDR_SITELOCAL(addr) (const struct in6_addr *)
490 #define IN6_IS_ADDR_V4MAPPED(addr) (const struct in6_addr *)
491 #define IN6_IS_ADDR_V4COMPAT(addr) (const struct in6_addr *)
492 #define IN6_IS_ADDR_MC_RESERVED(addr) (const struct in6_addr *)
493 #define IN6_IS_ADDR_MC_NODELOCAL(addr) (const struct in6_addr *)
494 #define IN6_IS_ADDR_MC_LINKLOCAL(addr) (const struct in6_addr *)
495 #define IN6_IS_ADDR_MC_SITELOCAL(addr) (const struct in6_addr *)
496 #define IN6_IS_ADDR_MC_ORGLLOCAL(addr) (const struct in6_addr *)
497 #define IN6_IS_ADDR_MC_GLOBAL(addr) (const struct in6_addr *)
498 #define IN6_IS_ADDR_6TO4(addr) (const struct in6_addr *)
499 #define IN6_IS_ADDR_6TO4_PREFIX_EQUAL(addr) (const struct in6_addr *)
500 #define IN6_IS_ADDR_LINKSCOPE(addr) (const struct in6_addr *)
501 #define IN6_IS_ADDR_UNSPECIFIED(addr) \
502 ((addr)->_S6_un._S6_u32[3] == 0) && \
503 ((addr)->_S6_un._S6_u32[2] == 0) && \
504 ((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] & 0x0000c0ff) == 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] & 0x0000c0ff) == 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))
597 #else /* _BIG_ENDIAN */
598 #define IN6_IS_ADDR_V4COMPAT(addr) \
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)
616 #else /* _BIG_ENDIAN */
617 #define IN6_IS_ADDR_MC_RESERVED(addr) \
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 ff02:0:0:0:0:1:ff00::/104.
667  */
668 #ifdef _BIG_ENDIAN
669 #define IN6_IS_ADDR_MC_SOLICITEDNODE(addr) \
670     (((addr)->_S6_un._S6_u32[0] == 0xff020000) && \
671      ((addr)->_S6_un._S6_u32[1] == 0x00000000) && \
672      ((addr)->_S6_un._S6_u32[2] == 0x00000001) && \
673      ((addr)->_S6_un._S6_u32[3] & 0xff000000) == 0xff000000))
674 #else
675 #define IN6_IS_ADDR_MC_SOLICITEDNODE(addr) \
676     (((addr)->_S6_un._S6_u32[0] == 0x000002ff) && \
677      ((addr)->_S6_un._S6_u32[1] == 0x00000000) && \
678      ((addr)->_S6_un._S6_u32[2] == 0x01000000) && \
679      ((addr)->_S6_un._S6_u32[3] & 0x000000ff) == 0x000000ff))
680 #endif

682 /*
683  * Macros to a) test for 6to4 IPv6 address, and b) to test if two
684  * 6to4 addresses have the same /48 prefix, and, hence, are from the
685  * same 6to4 site.
686  */

688 #ifdef _BIG_ENDIAN
689 #define IN6_IS_ADDR_6TO4(addr) \
690     (((addr)->_S6_un._S6_u32[0] & 0xffff0000) == 0x20020000)
691 #else /* _BIG_ENDIAN */
692 #define IN6_IS_ADDR_6TO4(addr) \
693     (((addr)->_S6_un._S6_u32[0] & 0x0000ffff) == 0x00000220)
694 #endif /* _BIG_ENDIAN */

696 #define IN6_ARE_6TO4_PREFIX_EQUAL(addr1, addr2) \
697     (((addr1)->_S6_un._S6_u32[0] == (addr2)->_S6_un._S6_u32[0]) && \
698      ((addr1)->_S6_un._S6_u8[4] == (addr2)->_S6_un._S6_u8[4]) && \
699      ((addr1)->_S6_un._S6_u8[5] == (addr2)->_S6_un._S6_u8[5]))

701 /*
702  * IN6_IS_ADDR_LINKSCOPE
703  * Identifies an address as being either link-local, link-local multicast or
704  * node-local multicast. All types of addresses are considered to be unique
705  * within the scope of a given link.
706  */
707 #define IN6_IS_ADDR_LINKSCOPE(addr) \
708     (IN6_IS_ADDR_LINKLOCAL(addr) || IN6_IS_ADDR_MC_LINKLOCAL(addr) || \
709      IN6_IS_ADDR_MC_NODELOCAL(addr))

711 /*
712  * Useful utility macros for operations with IPv6 addresses
713  * Note: These macros are NOT defined in the RFC2553 or any other
714  * standard specification and are not standard macros that portable
715  * applications should use.

```

```

716 */

718 /*
719  * IN6_V4MAPPED_TO_INADDR
720  * IN6_V4MAPPED_TO_IPADDR
721  * Assign a IPv4-Mapped IPv6 address to an IPv4 address.
722  * Note: These macros are NOT defined in RFC2553 or any other standard
723  * specification and are not macros that portable applications should
724  * use.
725  */
726 #define IN6_V4MAPPED_TO_INADDR(const in6_addr_t *v6, struct in_addr *v4);
727 #define 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 #define IN6_INADDR_TO_V4MAPPED(const struct in_addr *v4, in6_addr_t *v6);
745 #define 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)) ? 0xFFFFFFFFFu : \
857 (((qoctet) * 32) >= (prefix)) ? 0x00000000u : \
858 0xFFFFFFFFFu << (((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 *
883 * Note: Some of the IP_namespace has conflict with and
884 * and is exposed through <xti.h>. (It also requires exposing
885 * options not implemented). The options with potential
886 * for conflicts use #ifndef guards.
887 */
888 #ifndef IP_OPTIONS
889 #define IP_OPTIONS 1 /* set/get IP per-packet options */
890 #endif
891
892 #define IP_HDRINCL 2 /* int; header is included with data (raw) */
893
894 #ifndef IP_TOS
895 #define IP_TOS 3 /* int; IP type of service and precedence */
896 #endif
897
898 #ifndef IP_TTL
899 #define IP_TTL 4 /* int; IP time to live */
900 #endif
901
902 #define IP_RECVOPTS 0x5 /* int; receive all IP options w/datagram */
903 #define IP_RECVRETOPTS 0x6 /* int; receive IP options for response */
904 #define IP_RECVSTADDR 0x7 /* int; receive IP dst addr w/datagram */
905 #define IP_RETOPTS 0x8 /* ip_opts; set/get IP per-packet options */
906 #define IP_RECVIF 0x9 /* int; receive the inbound interface index */
907 #define IP_RECVSLLA 0xa /* sockaddr_dl; get source link layer address */
908 #define IP_RECVTTL 0xb /* uint8_t; get TTL for inbound packet */
909
910 #define IP_MULTICAST_IF 0x10 /* set/get IP multicast interface */
911 #define IP_MULTICAST_TTL 0x11 /* set/get IP multicast timetolive */
912 #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 * T_IP_OPTIONS 0x107 - IP per-packet options
994 * T_IP_TOS      0x108 - IP per packet type of service
995 */

997 /*
998  * Default value constants for multicast attributes controlled by
999  * IP*_MULTICAST_LOOP and IP*_MULTICAST_{TTL,HOPS} options.
1000 */
1001 #define IP_DEFAULT_MULTICAST_TTL 1      /* normally limit m'casts to 1 hop */
1002 #define IP_DEFAULT_MULTICAST_LOOP 1     /* normally hear sends if a member */

1004 #if !defined(_XPG4_2) || defined(__EXTENSIONS__)
1005 /*
1006  * Argument structure for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.
1007  */
1008 struct ip_mreq {
1009     struct in_addr imr_multiaddr; /* IP multicast address of group */
1010     struct in_addr imr_interface; /* local IP address of interface */
1011 };

1013 /*
1014  * Argument structure for IP_BLOCK_SOURCE, IP_UNBLOCK_SOURCE,
1015  * IP_ADD_SOURCE_MEMBERSHIP, and IP_DROP_SOURCE_MEMBERSHIP.
1016  */
1017 struct ip_mreq_source {
1018     struct in_addr imr_multiaddr; /* IP address of group */
1019     struct in_addr imr_sourceaddr; /* IP address of source */
1020     struct in_addr imr_interface; /* IP address of interface */
1021 };

1023 /*
1024  * Argument structure for IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP on
1025  * IPv6 addresses.
1026  */
1027 struct ipv6_mreq {
1028     struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast addr */
1029     unsigned int    ipv6mr_interface; /* interface index */
1030 };

1032 /*
1033  * Use #pragma pack() construct to force 32-bit alignment on amd64.
1034  * This is needed to keep the structure size and offsets consistent
1035  * between a 32-bit app and the 64-bit amd64 kernel in structures
1036  * where 64-bit alignment would create gaps (in this case, structures
1037  * which have a uint32_t followed by a struct sockaddr_storage).
1038  */
1039 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1040 #pragma pack(4)
1041 #endif

1043 /*
1044  * Argument structure for MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP.
1045  */

```

```

1046 struct group_req {
1047     uint32_t          gr_interface; /* interface index */
1048     struct sockaddr_storage gr_group; /* group address */
1049 };

1051 /*
1052  * Argument structure for MCAST_BLOCK_SOURCE, MCAST_UNBLOCK_SOURCE,
1053  * MCAST_JOIN_SOURCE_GROUP, MCAST_LEAVE_SOURCE_GROUP.
1054  */
1055 struct group_source_req {
1056     uint32_t          gsr_interface; /* interface index */
1057     struct sockaddr_storage gsr_group; /* group address */
1058     struct sockaddr_storage gsr_source; /* source address */
1059 };

1061 /*
1062  * Argument for SIOC[GS]MSFILTER ioctls
1063  */
1064 struct group_filter {
1065     uint32_t          gf_interface; /* interface index */
1066     struct sockaddr_storage gf_group; /* multicast address */
1067     uint32_t          gf_fmode; /* filter mode */
1068     uint32_t          gf_numsrc; /* number of sources */
1069     struct sockaddr_storage gf_slist[1]; /* source address */
1070 };

1072 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1073 #pragma pack()
1074 #endif

1076 #define GROUP_FILTER_SIZE(numsrc) \
1077     (sizeof (struct group_filter) - sizeof (struct sockaddr_storage) \
1078      + (numsrc) * sizeof (struct sockaddr_storage))

1080 /*
1081  * Argument for SIOC[GS]IPMSFILTER ioctls (IPv4-specific)
1082  */
1083 struct ip_msfilter {
1084     struct in_addr imsf_multiaddr; /* IP multicast address of group */
1085     struct in_addr imsf_interface; /* local IP address of interface */
1086     uint32_t imsf_fmode; /* filter mode */
1087     uint32_t imsf_numsrc; /* number of sources in src_list */
1088     struct in_addr imsf_slist[1]; /* start of source list */
1089 };

1091 #define IP_MSFILTER_SIZE(numsrc) \
1092     (sizeof (struct ip_msfilter) - sizeof (struct in_addr) \
1093      + (numsrc) * sizeof (struct in_addr))

1095 /*
1096  * Multicast source filter manipulation functions in libsocket;
1097  * defined in RFC 3678.
1098  */
1099 int setsourcefilter(int, uint32_t, struct sockaddr *, socklen_t, uint32_t,
1100                    uint_t, struct sockaddr_storage *);

1102 int getsourcefilter(int, uint32_t, struct sockaddr *, socklen_t, uint32_t *,
1103                    uint_t *, struct sockaddr_storage *);

1105 int setipv4sourcefilter(int, struct in_addr, struct in_addr, uint32_t,
1106                        uint32_t, struct in_addr *);

1108 int getipv4sourcefilter(int, struct in_addr, struct in_addr, uint32_t *,
1109                        uint32_t *, struct in_addr *);

1111 /*

```

```

1112 * Definitions needed for [gs]setsourcefilter(), [gs]setipv4sourcefilter()
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 */
1208
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 */
1222
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
1228
1229 #define IPV6_RECVRTHDR          0x16   /* enable/disable IPV6_RTHDR */
1230
1231 /*
1232  * enable/disable IPV6_RTHDRDSTOPTS. Now obsolete. IPV6_RECVDSTOPTS enables
1233  * the receipt of both headers.
1234  */
1235 #define IPV6_RECVRTHDRDSTOPTS  0x17
1236
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 */
1247
1248 /*
1249  * enable/disable receipt of both both IPV6_DSTOPTS headers.
1250  */
1251 #define IPV6_RECVDSTOPTS       0x28
1252
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 */
1262
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
1270
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
1277
1278 #define IPV6_PREFER_SRC_MASK (IPV6_PREFER_SRC_MIPMASK | \
1279     IPV6_PREFER_SRC_TMPMASK | IPV6_PREFER_SRC_CGAMASK)
1280
1281 #define IPV6_PREFER_SRC_DEFAULT (IPV6_PREFER_SRC_MIPDEFAULT | \
1282     IPV6_PREFER_SRC_TMPDEFAULT | IPV6_PREFER_SRC_CGADEFAULT)
1283
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) */
1290
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 */
1299
1300 #endif /* !defined(XPG4_2) || defined(__EXTENSIONS__) */
1301
1302 /*
1303  * Extern declarations for pre-defined global const variables
1304  */
1305 #if !defined(XPG4_2) || defined(__EXTENSIONS__)
1306 #ifndef _KERNEL
1307 #ifndef _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 Sat Aug 18 10:37:23 2012
new/usr/src/uts/common/sys/netstack.h
dccp: starting module template
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 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         } nu_s;
163     } netstack_u;
164     #define netstack_modules netstack_u.nu_modules
165     #define netstack_dls netstack_u.nu_s.nu_dls
166     #define netstack_iptun netstack_u.nu_s.nu_iptun
167     #define netstack_str netstack_u.nu_s.nu_str
168     #define netstack_hook netstack_u.nu_s.nu_hook
169     #define netstack_neti netstack_u.nu_s.nu_neti
170     #define netstack_arp netstack_u.nu_s.nu_arp
171     #define netstack_ip netstack_u.nu_s.nu_ip
172     #define netstack_icmp netstack_u.nu_s.nu_icmp
173     #define netstack_udp netstack_u.nu_s.nu_udp
174     #define netstack_tcp netstack_u.nu_s.nu_tcp
175     #define netstack_sctp netstack_u.nu_s.nu_sctp
176     #define netstack_rts netstack_u.nu_s.nu_rts
177     #define netstack_ipsec netstack_u.nu_s.nu_ipsec
178     #define netstack_keysock netstack_u.nu_s.nu_keysock
179     #define netstack_spdsock netstack_u.nu_s.nu_spdsock
180     #define netstack_ipsecah netstack_u.nu_s.nu_ipsecah
181     #define netstack_ipsecesp netstack_u.nu_s.nu_ipsecesp
182     #define netstack_ipnet netstack_u.nu_s.nu_ipnet
183     #define netstack_ilb netstack_u.nu_s.nu_ilb
184     #define netstack_dccp netstack_u.nu_s.nu_dccp
185     #endif /* ! codereview */
186
188     nm_state_t netstack_m_state[NS_MAX]; /* module state */
189
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 */
196
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;
203
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 */
210
211 #define NSF_ZONE_INPROGRESS \
212     (NSF_ZONE_CREATE|NSF_ZONE_SHUTDOWN|NSF_ZONE_DESTROY)
213
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 };
223
224 /* nr_flags values */
225 #define NRF_REGISTERED 0x01
226 #define NRF_DYING 0x02 /* No new creates */
227
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  */
234
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);
241
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);
245
246 extern netstack_t *netstack_get_current(void);
247
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/common/sys/sdt.h

1

```
*****
16746 Sat Aug 18 10:37:23 2012
new/usr/src/uts/common/sys/sdt.h
dccp: connect
*****
_____unchanged_portion_omitted_____

141 #define DTRACE_SCHED(name) \
142     DTRACE_PROBE(__sched_##name);

144 #define DTRACE_SCHED1(name, type1, arg1) \
145     DTRACE_PROBE1(__sched_##name, type1, arg1);

147 #define DTRACE_SCHED2(name, type1, arg1, type2, arg2) \
148     DTRACE_PROBE2(__sched_##name, type1, arg1, type2, arg2);

150 #define DTRACE_SCHED3(name, type1, arg1, type2, arg2, type3, arg3) \
151     DTRACE_PROBE3(__sched_##name, type1, arg1, type2, arg2, type3, arg3);

153 #define DTRACE_SCHED4(name, type1, arg1, type2, arg2, \
154     type3, arg3, type4, arg4) \
155     DTRACE_PROBE4(__sched_##name, type1, arg1, type2, arg2, \
156     type3, arg3, type4, arg4);

158 #define DTRACE_PROC(name) \
159     DTRACE_PROBE(__proc_##name);

161 #define DTRACE_PROC1(name, type1, arg1) \
162     DTRACE_PROBE1(__proc_##name, type1, arg1);

164 #define DTRACE_PROC2(name, type1, arg1, type2, arg2) \
165     DTRACE_PROBE2(__proc_##name, type1, arg1, type2, arg2);

167 #define DTRACE_PROC3(name, type1, arg1, type2, arg2, type3, arg3) \
168     DTRACE_PROBE3(__proc_##name, type1, arg1, type2, arg2, type3, arg3);

170 #define DTRACE_PROC4(name, type1, arg1, type2, arg2, \
171     type3, arg3, type4, arg4) \
172     DTRACE_PROBE4(__proc_##name, type1, arg1, type2, arg2, \
173     type3, arg3, type4, arg4);

175 #define DTRACE_IO(name) \
176     DTRACE_PROBE(__io_##name);

178 #define DTRACE_IO1(name, type1, arg1) \
179     DTRACE_PROBE1(__io_##name, type1, arg1);

181 #define DTRACE_IO2(name, type1, arg1, type2, arg2) \
182     DTRACE_PROBE2(__io_##name, type1, arg1, type2, arg2);

184 #define DTRACE_IO3(name, type1, arg1, type2, arg2, type3, arg3) \
185     DTRACE_PROBE3(__io_##name, type1, arg1, type2, arg2, type3, arg3);

187 #define DTRACE_IO4(name, type1, arg1, type2, arg2, \
188     type3, arg3, type4, arg4) \
189     DTRACE_PROBE4(__io_##name, type1, arg1, type2, arg2, \
190     type3, arg3, type4, arg4);

192 #define DTRACE_ISCSI_2(name, type1, arg1, type2, arg2) \
193     DTRACE_PROBE2(__iscsi_##name, type1, arg1, type2, arg2);

195 #define DTRACE_ISCSI_3(name, type1, arg1, type2, arg2, type3, arg3) \
196     DTRACE_PROBE3(__iscsi_##name, type1, arg1, type2, arg2, type3, arg3);

198 #define DTRACE_ISCSI_4(name, type1, arg1, type2, arg2, \
199     type3, arg3, type4, arg4) \
```

new/usr/src/uts/common/sys/sdt.h

2

```
200     DTRACE_PROBE4(__iscsi_##name, type1, arg1, type2, arg2, \
201     type3, arg3, type4, arg4);

203 #define DTRACE_ISCSI_5(name, type1, arg1, type2, arg2, \
204     type3, arg3, type4, arg4, type5, arg5) \
205     DTRACE_PROBE5(__iscsi_##name, type1, arg1, type2, arg2, \
206     type3, arg3, type4, arg4, type5, arg5);

208 #define DTRACE_ISCSI_6(name, type1, arg1, type2, arg2, \
209     type3, arg3, type4, arg4, type5, arg5, type6, arg6) \
210     DTRACE_PROBE6(__iscsi_##name, type1, arg1, type2, arg2, \
211     type3, arg3, type4, arg4, type5, arg5, type6, arg6);

213 #define DTRACE_ISCSI_7(name, type1, arg1, type2, arg2, \
214     type3, arg3, type4, arg4, type5, arg5, type6, arg6, type7, arg7) \
215     DTRACE_PROBE7(__iscsi_##name, type1, arg1, type2, arg2, \
216     type3, arg3, type4, arg4, type5, arg5, type6, arg6, \
217     type7, arg7);

219 #define DTRACE_ISCSI_8(name, type1, arg1, type2, arg2, \
220     type3, arg3, type4, arg4, type5, arg5, type6, arg6, \
221     type7, arg7, type8, arg8) \
222     DTRACE_PROBE8(__iscsi_##name, type1, arg1, type2, arg2, \
223     type3, arg3, type4, arg4, type5, arg5, type6, arg6, \
224     type7, arg7, type8, arg8);

226 #define DTRACE_NFSV3_3(name, type1, arg1, type2, arg2, \
227     type3, arg3) \
228     DTRACE_PROBE3(__nfsv3_##name, type1, arg1, type2, arg2, \
229     type3, arg3);

230 #define DTRACE_NFSV3_4(name, type1, arg1, type2, arg2, \
231     type3, arg3, type4, arg4) \
232     DTRACE_PROBE4(__nfsv3_##name, type1, arg1, type2, arg2, \
233     type3, arg3, type4, arg4);

235 #define DTRACE_NFSV4_1(name, type1, arg1) \
236     DTRACE_PROBE1(__nfsv4_##name, type1, arg1);

238 #define DTRACE_NFSV4_2(name, type1, arg1, type2, arg2) \
239     DTRACE_PROBE2(__nfsv4_##name, type1, arg1, type2, arg2);

241 #define DTRACE_NFSV4_3(name, type1, arg1, type2, arg2, type3, arg3) \
242     DTRACE_PROBE3(__nfsv4_##name, type1, arg1, type2, arg2, type3, arg3);

244 #define DTRACE_SMB_1(name, type1, arg1) \
245     DTRACE_PROBE1(__smb_##name, type1, arg1);

247 #define DTRACE_SMB_2(name, type1, arg1, type2, arg2) \
248     DTRACE_PROBE2(__smb_##name, type1, arg1, type2, arg2);

250 #define DTRACE_IP(name) \
251     DTRACE_PROBE(__ip_##name);

253 #define DTRACE_IP1(name, type1, arg1) \
254     DTRACE_PROBE1(__ip_##name, type1, arg1);

256 #define DTRACE_IP2(name, type1, arg1, type2, arg2) \
257     DTRACE_PROBE2(__ip_##name, type1, arg1, type2, arg2);

259 #define DTRACE_IP3(name, type1, arg1, type2, arg2, type3, arg3) \
260     DTRACE_PROBE3(__ip_##name, type1, arg1, type2, arg2, type3, arg3);

262 #define DTRACE_IP4(name, type1, arg1, type2, arg2, \
263     type3, arg3, type4, arg4) \
264     DTRACE_PROBE4(__ip_##name, type1, arg1, type2, arg2, \
265     type3, arg3, type4, arg4);
```

```

267 #define DTRACE_IP5(name, type1, arg1, type2, arg2, \
268     type3, arg3, type4, arg4, type5, arg5) \
269     DTRACE_PROBE5(__ip_##name, type1, arg1, type2, arg2, \
270     type3, arg3, type4, arg4, type5, arg5);

272 #define DTRACE_IP6(name, type1, arg1, type2, arg2, \
273     type3, arg3, type4, arg4, type5, arg5, type6, arg6) \
274     DTRACE_PROBE6(__ip_##name, type1, arg1, type2, arg2, \
275     type3, arg3, type4, arg4, type5, arg5, type6, arg6);

277 #define DTRACE_IP7(name, type1, arg1, type2, arg2, type3, arg3, \
278     type4, arg4, type5, arg5, type6, arg6, type7, arg7) \
279     DTRACE_PROBE7(__ip_##name, type1, arg1, type2, arg2, \
280     type3, arg3, type4, arg4, type5, arg5, type6, arg6, \
281     type7, arg7);

283 #define DTRACE_TCP(name) \
284     DTRACE_PROBE(__tcp_##name);

286 #define DTRACE_TCP1(name, type1, arg1) \
287     DTRACE_PROBE1(__tcp_##name, type1, arg1);

289 #define DTRACE_TCP2(name, type1, arg1, type2, arg2) \
290     DTRACE_PROBE2(__tcp_##name, type1, arg1, type2, arg2);

292 #define DTRACE_TCP3(name, type1, arg1, type2, arg2, type3, arg3) \
293     DTRACE_PROBE3(__tcp_##name, type1, arg1, type2, arg2, type3, arg3);

295 #define DTRACE_TCP4(name, type1, arg1, type2, arg2, \
296     type3, arg3, type4, arg4) \
297     DTRACE_PROBE4(__tcp_##name, type1, arg1, type2, arg2, \
298     type3, arg3, type4, arg4);

300 #define DTRACE_TCP5(name, type1, arg1, type2, arg2, \
301     type3, arg3, type4, arg4, type5, arg5) \
302     DTRACE_PROBE5(__tcp_##name, type1, arg1, type2, arg2, \
303     type3, arg3, type4, arg4, type5, arg5);

305 #define DTRACE_TCP6(name, type1, arg1, type2, arg2, \
306     type3, arg3, type4, arg4, type5, arg5, type6, arg6) \
307     DTRACE_PROBE6(__tcp_##name, type1, arg1, type2, arg2, \
308     type3, arg3, type4, arg4, type5, arg5, type6, arg6);

310 #define DTRACE_UDP(name) \
311     DTRACE_PROBE(__udp_##name);

313 #define DTRACE_UDP1(name, type1, arg1) \
314     DTRACE_PROBE1(__udp_##name, type1, arg1);

316 #define DTRACE_UDP2(name, type1, arg1, type2, arg2) \
317     DTRACE_PROBE2(__udp_##name, type1, arg1, type2, arg2);

319 #define DTRACE_UDP3(name, type1, arg1, type2, arg2, type3, arg3) \
320     DTRACE_PROBE3(__udp_##name, type1, arg1, type2, arg2, type3, arg3);

322 #define DTRACE_UDP4(name, type1, arg1, type2, arg2, \
323     type3, arg3, type4, arg4) \
324     DTRACE_PROBE4(__udp_##name, type1, arg1, type2, arg2, \
325     type3, arg3, type4, arg4);

327 #define DTRACE_UDP5(name, type1, arg1, type2, arg2, \
328     type3, arg3, type4, arg4, type5, arg5) \
329     DTRACE_PROBE5(__udp_##name, type1, arg1, type2, arg2, \
330     type3, arg3, type4, arg4, type5, arg5);

```

```

332 #define DTRACE_DCCP(name) \
333     DTRACE_PROBE(__dccp_##name);

335 #define DTRACE_DCCP1(name, type1, arg1) \
336     DTRACE_PROBE1(__dccp_##name, type1, arg1);

338 #define DTRACE_DCCP2(name, type1, arg1, type2, arg2) \
339     DTRACE_PROBE2(__dccp_##name, type1, arg1, type2, arg2);

341 #define DTRACE_DCCP3(name, type1, arg1, type2, arg2, type3, arg3) \
342     DTRACE_PROBE3(__dccp_##name, type1, arg1, type2, arg2, type3, arg3);

344 #define DTRACE_DCCP4(name, type1, arg1, type2, arg2, \
345     type3, arg3, type4, arg4) \
346     DTRACE_PROBE4(__dccp_##name, type1, arg1, type2, arg2, \
347     type3, arg3, type4, arg4);

349 #define DTRACE_DCCP5(name, type1, arg1, type2, arg2, \
350     type3, arg3, type4, arg4, type5, arg5) \
351     DTRACE_PROBE5(__dccp_##name, type1, arg1, type2, arg2, \
352     type3, arg3, type4, arg4, type5, arg5);

354 #define DTRACE_DCCP6(name, type1, arg1, type2, arg2, \
355     type3, arg3, type4, arg4, type5, arg5, type6, arg6) \
356     DTRACE_PROBE6(__dccp_##name, type1, arg1, type2, arg2, \
357     type3, arg3, type4, arg4, type5, arg5, type6, arg6);

359 #endif /* ! codereview */

361 #define DTRACE_SYSEVENT2(name, type1, arg1, type2, arg2) \
362     DTRACE_PROBE2(__sysevent_##name, type1, arg1, type2, arg2);

364 #define DTRACE_XPV(name) \
365     DTRACE_PROBE(__xpv_##name);

367 #define DTRACE_XPV1(name, type1, arg1) \
368     DTRACE_PROBE1(__xpv_##name, type1, arg1);

370 #define DTRACE_XPV2(name, type1, arg1, type2, arg2) \
371     DTRACE_PROBE2(__xpv_##name, type1, arg1, type2, arg2);

373 #define DTRACE_XPV3(name, type1, arg1, type2, arg2, type3, arg3) \
374     DTRACE_PROBE3(__xpv_##name, type1, arg1, type2, arg2, type3, arg3);

376 #define DTRACE_XPV4(name, type1, arg1, type2, arg2, type3, arg3, \
377     type4, arg4) \
378     DTRACE_PROBE4(__xpv_##name, type1, arg1, type2, arg2, \
379     type3, arg3, type4, arg4);

381 #define DTRACE_FC_1(name, type1, arg1) \
382     DTRACE_PROBE1(__fc_##name, type1, arg1);

384 #define DTRACE_FC_2(name, type1, arg1, type2, arg2) \
385     DTRACE_PROBE2(__fc_##name, type1, arg1, type2, arg2);

387 #define DTRACE_FC_3(name, type1, arg1, type2, arg2, type3, arg3) \
388     DTRACE_PROBE3(__fc_##name, type1, arg1, type2, arg2, type3, arg3);

390 #define DTRACE_FC_4(name, type1, arg1, type2, arg2, type3, arg3, type4, arg4) \
391     DTRACE_PROBE4(__fc_##name, type1, arg1, type2, arg2, type3, arg3, \
392     type4, arg4);

394 #define DTRACE_FC_5(name, type1, arg1, type2, arg2, type3, arg3, \
395     type4, arg4, type5, arg5) \
396     DTRACE_PROBE5(__fc_##name, type1, arg1, type2, arg2, type3, arg3, \
397     type4, arg4, type5, arg5);

```

```
399 #define DTRACE_SRP_1(name, type1, arg1)          \
400     DTRACE_PROBE1(__srp_#name, type1, arg1);

402 #define DTRACE_SRP_2(name, type1, arg1, type2, arg2) \
403     DTRACE_PROBE2(__srp_#name, type1, arg1, type2, arg2);

405 #define DTRACE_SRP_3(name, type1, arg1, type2, arg2, type3, arg3) \
406     DTRACE_PROBE3(__srp_#name, type1, arg1, type2, arg2, type3, arg3);

408 #define DTRACE_SRP_4(name, type1, arg1, type2, arg2, type3, arg3, \
409     type4, arg4) \
410     DTRACE_PROBE4(__srp_#name, type1, arg1, type2, arg2, \
411     type3, arg3, type4, arg4);

413 #define DTRACE_SRP_5(name, type1, arg1, type2, arg2, type3, arg3, \
414     type4, arg4, type5, arg5) \
415     DTRACE_PROBE5(__srp_#name, type1, arg1, type2, arg2, \
416     type3, arg3, type4, arg4, type5, arg5);

418 #define DTRACE_SRP_6(name, type1, arg1, type2, arg2, type3, arg3, \
419     type4, arg4, type5, arg5, type6, arg6) \
420     DTRACE_PROBE6(__srp_#name, type1, arg1, type2, arg2, \
421     type3, arg3, type4, arg4, type5, arg5, type6, arg6);

423 #define DTRACE_SRP_7(name, type1, arg1, type2, arg2, type3, arg3, \
424     type4, arg4, type5, arg5, type6, arg6, type7, arg7) \
425     DTRACE_PROBE7(__srp_#name, type1, arg1, type2, arg2, \
426     type3, arg3, type4, arg4, type5, arg5, type6, arg6, type7, arg7);

428 #define DTRACE_SRP_8(name, type1, arg1, type2, arg2, type3, arg3, \
429     type4, arg4, type5, arg5, type6, arg6, type7, arg7, type8, arg8) \
430     DTRACE_PROBE8(__srp_#name, type1, arg1, type2, arg2, \
431     type3, arg3, type4, arg4, type5, arg5, type6, arg6, \
432     type7, arg7, type8, arg8);

434 #endif /* _KERNEL */

436 extern const char *sdt_prefix;

438 typedef struct sdt_probedesc {
439     char          *sdpd_name;      /* name of this probe */
440     unsigned long sdpd_offset;     /* offset of call in text */
441     struct sdt_probedesc *sdpd_next; /* next static probe */
442 } sdt_probedesc_t;

444 #ifdef __cplusplus
445 }
446 #endif

448 #endif /* _SYS_SDT_H */
```


new/usr/src/uts/intel/Makefile.intel.shared

1

```
*****
16840 Sat Aug 18 10:37:24 2012
new/usr/src/uts/intel/Makefile.intel.shared
dccc: starting module template
*****

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

22 # 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 = $(OBS_DIR)
50 LINT_LIB_DIR = $(UTSBASE)/intel/lint-libs/$(OBS_DIR)

52 UNIX_O = $(UNIX_DIR)/$(OBS_DIR)/unix.o
53 GENLIB = $(GENLIB_DIR)/$(OBS_DIR)/libgenunix.so
54 MODSTUBS_O = $(MODSTUBS_DIR)/$(OBS_DIR)/modstubs.o
55 LINT_LIB = $(UTSBASE)/i86pc/lint-libs/$(OBS_DIR)/llib-lunix.ln
56 GEN_LINT_LIB = $(UTSBASE)/intel/lint-libs/$(OBS_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)
```

new/usr/src/uts/intel/Makefile.intel.shared

3

```

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))$(OBS_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

```

new/usr/src/uts/intel/Makefile.intel.shared

4

```

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        += audioemul0k
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        += ntnx
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 DRV_KMODS        += dccp6
235 #endif /* ! codereview */
236 DRV_KMODS        += devinfo
237 DRV_KMODS        += dld
238 DRV_KMODS        += dlpistub
239 DRV_KMODS_32     += dnet
240 DRV_KMODS        += dump
241 DRV_KMODS        += ecpp
242 DRV_KMODS        += emlxs
243 DRV_KMODS        += fd
244 DRV_KMODS        += fdc
245 DRV_KMODS        += fm
246 DRV_KMODS        += fssnap
247 DRV_KMODS        += hxge
248 DRV_KMODS        += i8042
249 DRV_KMODS        += i915
250 DRV_KMODS        += icmp
251 DRV_KMODS        += icmp6
252 DRV_KMODS        += intel_nb5000
253 DRV_KMODS        += intel_nhm
254 DRV_KMODS        += ip
255 DRV_KMODS        += ip6
256 DRV_KMODS        += ipf
257 DRV_KMODS        += ipnet
258 DRV_KMODS        += ipnetl
259 DRV_KMODS        += ipsecsh

```

```

260 DRV_KMODS      += ipsecesp
261 DRV_KMODS      += ipw
262 DRV_KMODS      += iwh
263 DRV_KMODS      += iwi
264 DRV_KMODS      += iwk
265 DRV_KMODS      += iwp
266 DRV_KMODS      += iwscn
267 DRV_KMODS      += kb8042
268 DRV_KMODS      += keysock
269 DRV_KMODS      += kssl
270 DRV_KMODS      += kstat
271 DRV_KMODS      += ksyms
272 DRV_KMODS      += kmdb
273 DRV_KMODS      += llcl
274 DRV_KMODS      += lofi
275 DRV_KMODS      += log
276 DRV_KMODS      += loginmux
277 DRV_KMODS      += mega_sas
278 DRV_KMODS      += mc-amd
279 DRV_KMODS      += mm
280 DRV_KMODS      += mouse8042
281 DRV_KMODS      += mpt_sas
282 DRV_KMODS      += mr_sas
283 DRV_KMODS      += mwl
284 DRV_KMODS      += nca
285 DRV_KMODS      += nsmb
286 DRV_KMODS      += nulldriver
287 DRV_KMODS      += nv_sata
288 DRV_KMODS      += nxge
289 DRV_KMODS      += oce
290 DRV_KMODS      += openeep
291 DRV_KMODS      += pci_pci
292 DRV_KMODS      += pcic
293 DRV_KMODS      += pcieb
294 DRV_KMODS      += physmem
295 DRV_KMODS      += pcan
296 DRV_KMODS      += pcwl
297 DRV_KMODS      += pit_beep
298 DRV_KMODS      += pm
299 DRV_KMODS      += poll
300 DRV_KMODS      += pool
301 DRV_KMODS      += power
302 DRV_KMODS      += pseudo
303 DRV_KMODS      += ptc
304 DRV_KMODS      += ptm
305 DRV_KMODS      += pts
306 DRV_KMODS      += ptsl
307 DRV_KMODS      += qlge
308 DRV_KMODS      += radeon
309 DRV_KMODS      += ral
310 DRV_KMODS      += ramdisk
311 DRV_KMODS      += random
312 DRV_KMODS      += rds
313 DRV_KMODS      += rdsv3
314 DRV_KMODS      += rpcib
315 DRV_KMODS      += rsm
316 DRV_KMODS      += rts
317 DRV_KMODS      += rtw
318 DRV_KMODS      += rum
319 DRV_KMODS      += rwd
320 DRV_KMODS      += rwn
321 DRV_KMODS      += sad
322 DRV_KMODS      += sd
323 DRV_KMODS      += sdhost
324 DRV_KMODS      += sgen
325 DRV_KMODS      += si3124

```

```

326 DRV_KMODS      += smbios
327 DRV_KMODS      += softmac
328 DRV_KMODS      += spdssock
329 DRV_KMODS      += smbsrv
330 DRV_KMODS      += smp
331 DRV_KMODS      += spps
332 DRV_KMODS      += sppptun
333 DRV_KMODS      += srpt
334 DRV_KMODS      += st
335 DRV_KMODS      += sy
336 DRV_KMODS      += sysevent
337 DRV_KMODS      += sysmsg
338 DRV_KMODS      += tcp
339 DRV_KMODS      += tcp6
340 DRV_KMODS      += tl
341 DRV_KMODS      += tnf
342 DRV_KMODS      += tpm
343 DRV_KMODS      += trill
344 DRV_KMODS      += udp
345 DRV_KMODS      += udp6
346 DRV_KMODS      += ucode
347 DRV_KMODS      += ural
348 DRV_KMODS      += uath
349 DRV_KMODS      += urtw
350 DRV_KMODS      += vgatext
351 DRV_KMODS      += heci
352 DRV_KMODS      += vnica
353 DRV_KMODS      += vscan
354 DRV_KMODS      += wc
355 DRV_KMODS      += winlock
356 DRV_KMODS      += wpi
357 DRV_KMODS      += xge
358 DRV_KMODS      += yge
359 DRV_KMODS      += zcons
360 DRV_KMODS      += zyd
361 DRV_KMODS      += simnet
362 DRV_KMODS      += stmf
363 DRV_KMODS      += stmf_sbd
364 DRV_KMODS      += fct
365 DRV_KMODS      += fcoe
366 DRV_KMODS      += fcoet
367 DRV_KMODS      += fcoel
368 DRV_KMODS      += qlt
369 DRV_KMODS      += iscsit
370 DRV_KMODS      += pppt
371 DRV_KMODS      += ncall nsctl sdbc nsfern sv
372 DRV_KMODS      += ii rdc rdcsrv rdcsstub
373 DRV_KMODS      += iptun

375 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += bmc
376 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += glm
377 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += intel_nhmex
378 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += cpqary3
379 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += marvell188sx
380 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += bcm_sata
381 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += memtest
382 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += mpt
383 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += atiatom
384 $(CLOSED_BUILD)CLOSED_DRV_KMODS      += acpi_toshiba

386 #
387 # Common code drivers
388 #

390 DRV_KMODS      += afe
391 DRV_KMODS      += atge

```

```

392 DRV_KMODS      += bfe
393 DRV_KMODS      += dmfe
394 DRV_KMODS      += e1000g
395 DRV_KMODS      += efe
396 DRV_KMODS      += elx1
397 DRV_KMODS      += hme
398 DRV_KMODS      += mxfe
399 DRV_KMODS      += nge
400 DRV_KMODS      += pcn
401 DRV_KMODS      += rge
402 DRV_KMODS      += rtls
403 DRV_KMODS      += sfe
404 DRV_KMODS      += amd8111s
405 DRV_KMODS      += igb
406 DRV_KMODS      += iprb
407 DRV_KMODS      += ixgbe
408 DRV_KMODS      += vr
409 $(CLOSED_BUILD)CLOSED_DRV_KMODS += ixgb

```

```

411 #
412 #      DTrace and DTrace Providers
413 #
414 DRV_KMODS      += dtrace
415 DRV_KMODS      += fbt
416 DRV_KMODS      += lockstat
417 DRV_KMODS      += profile
418 DRV_KMODS      += sdt
419 DRV_KMODS      += systtrace
420 DRV_KMODS      += fasttrap
421 DRV_KMODS      += dcpc

```

```

423 #
424 #      I/O framework test drivers
425 #
426 DRV_KMODS      += pshot
427 DRV_KMODS      += gen_drv
428 DRV_KMODS      += tvhci tphci tclient
429 DRV_KMODS      += emul64

```

```

431 #
432 #      Machine Specific Driver Modules (/kernel/drv):
433 #
434 DRV_KMODS      += options
435 DRV_KMODS      += scsi_vhci
436 DRV_KMODS      += pmcs
437 DRV_KMODS      += pmcs8001fw
438 DRV_KMODS      += arcmsr
439 DRV_KMODS      += fcp
440 DRV_KMODS      += fcip
441 DRV_KMODS      += fcsn
442 DRV_KMODS      += fp
443 DRV_KMODS      += qlc
444 DRV_KMODS      += iscsi

```

```

446 #
447 #      PCMCIA specific module(s)
448 #
449 DRV_KMODS      += pcs
450 DRV_KMODS      += pcata
451 MISC_KMODS     += cardbus
452 $(CLOSED_BUILD)CLOSED_DRV_KMODS += pcser

```

```

454 #
455 #      SCSI Enclosure Services driver
456 #
457 DRV_KMODS      += ses

```

```

459 #
460 #      USB specific modules
461 #
462 DRV_KMODS      += hid
463 DRV_KMODS      += hwarc hwahc
464 DRV_KMODS      += hubd
465 DRV_KMODS      += uhci
466 DRV_KMODS      += ehci
467 DRV_KMODS      += ohci
468 DRV_KMODS      += usb_mid
469 DRV_KMODS      += usb_ia
470 DRV_KMODS      += scsa2usb
471 DRV_KMODS      += usbpri
472 DRV_KMODS      += ugen
473 DRV_KMODS      += usbser
474 DRV_KMODS      += usbsacm
475 DRV_KMODS      += usbsksp
476 DRV_KMODS      += usbsprl
477 DRV_KMODS      += usb_ac
478 DRV_KMODS      += usb_as
479 DRV_KMODS      += usbskel
480 DRV_KMODS      += usbvc
481 DRV_KMODS      += usbftdi
482 DRV_KMODS      += wusb_df
483 DRV_KMODS      += wusb_ca
484 DRV_KMODS      += usbecm

```

```

486 $(CLOSED_BUILD)CLOSED_DRV_KMODS += usbser_edge

```

```

488 #
489 #      1394 modules
490 #
491 MISC_KMODS     += s1394 sbp2
492 DRV_KMODS      += hci1394 scsa1394
493 DRV_KMODS      += av1394
494 DRV_KMODS      += dcaml394

```

```

496 #
497 #      InfiniBand pseudo drivers
498 #
499 DRV_KMODS      += ib ibp eibnx eoib rdsib sdp iser daplt hermon tavor sol_ucma
500 DRV_KMODS      += sol_umad

```

```

502 #
503 #      LVM modules
504 #
505 DRV_KMODS      += md
506 MISC_KMODS     += md_stripe md_hotspares md_mirror md_raid md_trans md_notify
507 MISC_KMODS     += md_sp

```

```

509 #
510 #      Brand modules
511 #
512 BRAND_KMODS    += snl_brand s10_brand

```

```

514 #
515 #      Exec Class Modules (/kernel/exec):
516 #
517 EXEC_KMODS     += elfexec intpexec shbinexec javaexec

```

```

519 #
520 #      Scheduling Class Modules (/kernel/sched):
521 #
522 SCHED_KMODS    += IA RT TS RT_DPTBL TS_DPTBL FSS FX FX_DPTBL SDC

```

```

524 #
525 #       File System Modules (/kernel/fs):
526 #
527 FS_KMODS      += autofs cacheefs ctfs dcfs dev devfs fdcs fifofs hsfcs lofs
528 FS_KMODS      += mntfs namefs nfs objfs zfs zut
529 FS_KMODS      += pcfs procfs sockfs specfs tmpfs udfs ufs sharefs
530 FS_KMODS      += smbfs

532 #
533 #       Streams Modules (/kernel/strmod):
534 #
535 STRMOD_KMODS  += bufmod connld dedump ldterm pckt pfmod pipemod
536 STRMOD_KMODS  += ptem redirmod rpcmod rlmod telmod timod
537 STRMOD_KMODS  += sppsasyn sppscomp
538 STRMOD_KMODS  += tirdwr ttcompat
539 STRMOD_KMODS  += usbkkm
540 STRMOD_KMODS  += usbms
541 STRMOD_KMODS  += usbwcm
542 STRMOD_KMODS  += usb_ah
543 STRMOD_KMODS  += drcompat
544 STRMOD_KMODS  += cryptmod
545 STRMOD_KMODS  += vuid2ps2
546 STRMOD_KMODS  += vuid3ps2
547 STRMOD_KMODS  += vuidm3p
548 STRMOD_KMODS  += vuidm4p
549 STRMOD_KMODS  += vuidm5p

551 #
552 #       'System' Modules (/kernel/sys):
553 #
554 SYS_KMODS     += c2audit
555 SYS_KMODS     += doorfs
556 SYS_KMODS     += exacsys
557 SYS_KMODS     += inst_sync
558 SYS_KMODS     += kaio
559 SYS_KMODS     += msgsys
560 SYS_KMODS     += pipe
561 SYS_KMODS     += portfs
562 SYS_KMODS     += pset
563 SYS_KMODS     += semsys
564 SYS_KMODS     += shmsys
565 SYS_KMODS     += sysacct
566 SYS_KMODS     += acctctl

568 #
569 #       'Misc' Modules (/kernel/misc)
570 #       MISC_KMODS are built both 32-bit and 64-bit
571 #       MISC_KMODS_32 are built only 32-bit
572 #       MISC_KMODS_64 are built only 64-bit
573 #
574 MISC_KMODS    += ac97
575 MISC_KMODS    += acpica
576 MISC_KMODS    += agpmaster
577 MISC_KMODS    += bignum
578 MISC_KMODS    += bootdev
579 MISC_KMODS    += busra
580 MISC_KMODS    += cmlb
581 MISC_KMODS    += consconfig
582 MISC_KMODS    += ctf
583 MISC_KMODS    += dadk
584 MISC_KMODS    += dcopy
585 MISC_KMODS    += dls
586 MISC_KMODS    += drm
587 MISC_KMODS    += fssnap_if
588 MISC_KMODS    += gda
589 MISC_KMODS    += gld

```

```

590 MISC_KMODS    += hidparser
591 MISC_KMODS    += hook
592 MISC_KMODS    += hpcsvc
593 MISC_KMODS    += ibcm
594 MISC_KMODS    += ibdm
595 MISC_KMODS    += ibdma
596 MISC_KMODS    += ibmf
597 MISC_KMODS    += ibt1
598 MISC_KMODS    += idm
599 MISC_KMODS    += idmap
600 MISC_KMODS    += iommulib
601 MISC_KMODS    += ipc
602 MISC_KMODS    += kbtrans
603 MISC_KMODS    += kcf
604 MISC_KMODS    += kgssapi
605 MISC_KMODS    += kmech_dummy
606 MISC_KMODS    += kmech_krb5
607 MISC_KMODS    += ksocket
608 MISC_KMODS    += mac
609 MISC_KMODS    += mii
610 MISC_KMODS    += mwlfw
611 MISC_KMODS    += net80211
612 MISC_KMODS    += nfs_d1boot
613 MISC_KMODS    += nfssrv
614 MISC_KMODS    += neti
615 MISC_KMODS    += pci_autoconfig
616 MISC_KMODS    += pcicfg
617 MISC_KMODS    += pcihp
618 MISC_KMODS    += pcmcia
619 MISC_KMODS    += rpcsec
620 MISC_KMODS    += rpcsec_gss
621 MISC_KMODS    += rsmops
622 MISC_KMODS    += sata
623 MISC_KMODS    += scsi
624 MISC_KMODS    += sda
625 MISC_KMODS    += sol_ofs
626 MISC_KMODS    += spuni
627 MISC_KMODS    += strategy
628 MISC_KMODS    += strplumb
629 MISC_KMODS    += tem
630 MISC_KMODS    += tlimod
631 MISC_KMODS    += usba usba10 usbs49_fw
632 MISC_KMODS    += scsi_vhci_f_sym_hds
633 MISC_KMODS    += scsi_vhci_f_sym
634 MISC_KMODS    += scsi_vhci_f_tpgs
635 MISC_KMODS    += scsi_vhci_f_asym_sun
636 MISC_KMODS    += scsi_vhci_f_tape
637 MISC_KMODS    += scsi_vhci_f_tpgs_tape
638 MISC_KMODS    += fctl
639 MISC_KMODS    += emlxs_fw
640 MISC_KMODS    += qlc_fw_2200
641 MISC_KMODS    += qlc_fw_2300
642 MISC_KMODS    += qlc_fw_2400
643 MISC_KMODS    += qlc_fw_2500
644 MISC_KMODS    += qlc_fw_6322
645 MISC_KMODS    += qlc_fw_8100
646 MISC_KMODS    += hwa1480_fw
647 MISC_KMODS    += uathfw
648 MISC_KMODS    += uwba

650 $(CLOSED_BUILD)CLOSED_MISC_KMODS      += klmmod klmops
651 $(CLOSED_BUILD)CLOSED_MISC_KMODS      += scsi_vhci_f_asym_lsi
652 $(CLOSED_BUILD)CLOSED_MISC_KMODS      += scsi_vhci_f_asym_emc
653 $(CLOSED_BUILD)CLOSED_MISC_KMODS      += scsi_vhci_f_sym_emc

655 #

```

```

656 #      Software Cryptographic Providers (/kernel/crypto):
657 #
658 CRYPTO_KMODS      += aes
659 CRYPTO_KMODS      += arcfour
660 CRYPTO_KMODS      += blowfish
661 CRYPTO_KMODS      += des
662 CRYPTO_KMODS      += ecc
663 CRYPTO_KMODS      += md4
664 CRYPTO_KMODS      += md5
665 CRYPTO_KMODS      += rsa
666 CRYPTO_KMODS      += sha1
667 CRYPTO_KMODS      += sha2
668 CRYPTO_KMODS      += swrand
669
670 #
671 #      IP Policy Modules (/kernel/ipp)
672 #
673 IPP_KMODS          += dlcosmk
674 IPP_KMODS          += flowacct
675 IPP_KMODS          += ipgpc
676 IPP_KMODS          += dscpmk
677 IPP_KMODS          += tokenmt
678 IPP_KMODS          += tswtclmt
679
680 #
681 #      generic-unix module (/kernel/genunix):
682 #
683 GENUNIX_KMODS      += genunix
684
685 #
686 #      SVVS Testing Modules (/kernel/strmod):
687 #
688 #      These are streams and driver modules which are not to be
689 #      delivered with a released system. However, during development
690 #      it is convenient to build and install the SVVS kernel modules.
691 #
692 SVVS_KMODS          += lmodb lmode lmodr lmodt svvslo tidg tivc tmux
693
694 $(CLOSED_BUILD)SVVS      += svvs
695
696 #
697 #      Modules eXcluded from the product:
698 #
699 $(CLOSED_BUILD)CLOSED_XMODS = \
700     adpu320          \
701     bnx              \
702     bnxe             \
703     lsimega          \
704     sdpiib
705
706 #
707 #      'Dacf' Modules (/kernel/dacf):
708 #
709 #
710 #
711 #
712 #      Performance Counter BackEnd modules (/usr/kernel/pcbe)
713 #
714 PCBE_KMODS          += pl23_pcbe p4_pcbe opteron_pcbe core_pcbe
715
716 #
717 #      MAC-Type Plugin Modules (/kernel/mac)
718 #
719 MAC_KMODS            += mac_6to4
720 MAC_KMODS            += mac_ether
721 MAC_KMODS            += mac_ipv4

```

```

722 MAC_KMODS           += mac_ipv6
723 MAC_KMODS           += mac_wifi
724 MAC_KMODS           += mac_ib
725
726 #
727 #      socketmod (kernel/socketmod)
728 #
729 SOCKET_KMODS        += sockpfp
730 SOCKET_KMODS        += socksctp
731 SOCKET_KMODS        += socksdp
732 SOCKET_KMODS        += sockrds
733 SOCKET_KMODS        += ksslf
734
735 #
736 #      kiconv modules (/kernel/kiconv):
737 #
738 KICONV_KMODS        += kiconv_emea kiconv_ja kiconv_ko kiconv_sc kiconv_tc
739
740 #
741 #      'Dacf' Modules (/kernel/dacf):
742 #
743 DACF_KMODS          += net_dacf

```

new/usr/src/uts/intel/dccp/Makefile

1

2348 Sat Aug 18 10:37:24 2012
new/usr/src/uts/intel/dccp/Makefile
dccp: starting with options

```
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 = $(DCCP_OBJS:%=$(OBJSDIR)/%)
42 LINTS = $(DCCP_OBJS:%.o=$(LINTSDIR)/%.ln)
43 ROOTMODDIR = $(ROOT_DRV_DIR)/$(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 = $(IP_CHECK_OBJS:%=../ip/$(OBJSDIR)/%)
53 #
54 #
55 # Include common rules.
56 #
57 include $(UTSBASE)/intel/Makefile.intel
58 #
59 #
60 # Define targets
61 #
```

new/usr/src/uts/intel/dccp/Makefile

2

```
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 */
```

new/usr/src/uts/intel/dccp6/Makefile

1

2178 Sat Aug 18 10:37:25 2012
new/usr/src/uts/intel/dccp6/Makefile
dccp: clean up

```
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 = ../../
35 #
36 #
37 # Define the module and object file sets.
38 #
39 MODULE = dccp6
40 OBJECTS = $(TCP6_OBJS:%=$(OBJDIR)/%)
41 LINTS = $(TCP6_OBJS:%.o=$(LINTDIR)/%.ln)
42 ROOTMODULE = $(ROOT_DRV_DIR)/$(MODULE)
43 CONF_SRCDIR = $(UTSBASE)/common/inet/dccp
44 #
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 = $(IP_CHECK_OBJS:%=../ip/$(OBJDIR)/%)
51 #
52 #
53 # Include common rules.
54 #
55 include $(UTSBASE)/intel/Makefile.intel
56 #
57 #
58 # Define targets
59 #
60 ALL_TARGET = $(BINARY) $(SRC_CONFFILE)
61 LINT_TARGET = $(MODULE).lint
```

new/usr/src/uts/intel/dccp6/Makefile

2

```
62 INSTALL_TARGET = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE)
63 #
64 #
65 # depends on tcp ip and ip6
66 #
67 LDFLAGS += -dy -Ndrv/tcp -Ndrv/ip -Ndrv/ip6
68 #
69 #
70 # Default build targets.
71 #
72 .KEEP_STATE:
73 #
74 def: $(DEF_DEPS)
75 #
76 all: $(ALL_DEPS)
77 #
78 clean: $(CLEAN_DEPS)
79 #
80 clobber: $(CLOBBER_DEPS)
81 #
82 lint: $(LINT_DEPS)
83 #
84 modlintlib: $(MODLINTLIB_DEPS)
85 #
86 clean.lint: $(CLEAN_LINT_DEPS)
87 #
88 install: $(INSTALL_DEPS)
89 #
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

6231 Sat Aug 18 10:37:25 2012
new/usr/src/uts/intel/ip/ip.global-objs.debug64
dccp: starting module template

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
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_addrs
46 cl_sctp_connect
47 cl_sctp_disconnect
48 cl_sctp_listen
49 cl_sctp_unlisten
50 conn_drain_nthreads
51 dccp_acceptor_rinit
52 dccp_acceptor_winit
53 dccp_bind_fanout_size
54 dccp_conn_cache
55 dccp_fallback_sock_winit
56 dccp_g_t_info_ack
57 dccp_g_t_info_ack_v6
58 dccp_max_optsize
59 dccp_opt_arr
60 dccp_opt_obj
61 dccp_propinfo_count
```

new/usr/src/uts/intel/ip/ip.global-objs.debug64

2

```
62 dccp_propinfo_tbl
63 dccp_random_anon_port
64 dccp_rinfo
65 dccp_rinitv4
66 dccp_rinitv6
67 dccp_sock_winit
68 dccp_squeue_flag
69 dccp_timercache
70 dccp_valid_levels_arr
71 dccp_winfo
72 dccp_winit
73 dccpinfov4
74 dccpinfov6
75 #endif /* ! codereview */
76 dce_cache
77 default_ip6_asp_table
78 do_tcp_fusion
79 do_tcpzcopy
80 dohwcksum
81 dummy_mod_info
82 dummymodinfo
83 dummyrmodinit
84 dummywmodinit
85 eventq_queue_in
86 eventq_queue_nic
87 eventq_queue_out
88 fsw
89 gcdb_hash
90 gcdb_hash_size
91 gcdb_lock
92 gcgrp4_hash
93 gcgrp6_hash
94 gcgrp_hash_size
95 gcgrp_lock
96 icmp_fallback_sock_winit
97 icmp_frag_size_table
98 icmp_g_t_info_ack
99 icmp_ipha
100 icmp_max_optsize
101 icmp_mod_info
102 icmp_opt_arr
103 icmp_opt_obj
104 icmp_propinfo_tbl
105 icmp_valid_levels_arr
106 icmpinfov4
107 icmpinfov6
108 icmprinitv4
109 icmprinitv6
110 icmpwinit
111 ilb_conn_cache
112 ilb_conn_cache_timeout
113 ilb_conn_hash_size
114 ilb_conn_tcp_expiry
115 ilb_conn_timer_size
116 ilb_conn_udp_expiry
117 ilb_kstat_instance
118 ilb_kmem_flags
119 ilb_nat_src_hash_size
120 ilb_nat_src_instance
121 ilb_rule_hash_size
122 ilb_sticky_cache
123 ilb_sticky_hash_size
124 ilb_sticky_expiry
125 ilb_sticky_timer_size
126 ilb_sticky_timeout
127 ill_no_arena
```

```

128 ill_null
129 inet_dev_info
130 inet_devops
131 ip6_fhtable_hash_size
132 ip6opt_ls
133 ip_cgtp_filter_rev
134 ip_conn_cache
135 ip_debug
136 ip_g_all_ones
137 ip_helper_stream_info
138 ip_helper_stream_rinit
139 ip_helper_stream_winit
140 ip_ioctl_ftbl
141 ip_loopback_mtu_v6plus
142 ip_loopback_mtuplus
143 ip_m_tbl
144 ip_max_frag_dups
145 ip_min_frag_prune_time
146 ip_minor_arena_la
147 ip_minor_arena_sa
148 ip_misc_ioctl_count
149 ip_misc_ioctl_table
150 ip_mod_info
151 ip_modclose_ackwait_ms
152 ip_ndx_ioctl_count
153 ip_ndx_ioctl_table
154 ip_poll_normal_ms
155 ip_poll_normal_ticks
156 ip_propinfo_tbl
157 ip_propinfo_count
158 ip_rput_pullups
159 ip_six_byte_all_ones
160 ip_squeue_create_callback
161 ip_squeue_enter
162 ip_squeue_fanout
163 ip_squeue_flag
164 ip_squeue_worker_wait
165 ip_thread_data
166 ip_thread_list
167 ip_thread_rwlock
168 ipcl_bind_fanout_size
169 ipcl_conn_hash_maxsize
170 ipcl_conn_hash_memfactor
171 ipcl_conn_hash_size
172 ipcl_dccp_bind_fanout_size
173 ipcl_dccp_conn_fanout_size
174 #endif /* !codereview */
175 ipcl_iptun_fanout_size
176 ipcl_raw_fanout_size
177 ipcl_udp_fanout_size
178 ipif_loopback_name
179 ipif_zero
180 ipinfov4
181 ipinfov6
182 iplrinit
183 iplwinit
184 ipmp_kstats
185 iprinitv4
186 iprinitv6
187 ipsec_action_cache
188 ipsec_hdr_pullup_needed
189 ipsec_pol_cache
190 ipsec_policy_failure_msgs
191 ipsec_sel_cache
192 ipsec_spd_hashsize
193 ipsec_weird_null_inbound_policy

```

```

194 ipv4info
195 ipv6_all_hosts_mcast
196 ipv6_all_ones
197 ipv6_all_rtrs_mcast
198 ipv6_all_v2rtrs_mcast
199 ipv6_all_zeros
200 ipv6_ll_template
201 ipv6_loopback
202 ipv6_solicited_node_mcast
203 ipv6_unspecified_group
204 ipv6info
205 ipwinit
206 ire_cache
207 ire_gw_secattr_cache
208 ire_null
209 ire_nv_arr
210 ire_nv_tbl
211 lcl_param_arr
212 mask_rnhead
213 max_keylen
214 modldrv
215 modlinkage
216 modlstrmod
217 multicast_encap_iphdr
218 nce_cache
219 ncec_cache
220 netdev_privs
221 prov_update_handle
222 radix_mask_cache
223 radix_node_cache
224 rawip_conn_cache
225 recvq_call
226 recvq_loop_cnt
227 req_arr
228 rinit_arp
229 rn_mkfreelist
230 rn_ones
231 rn_zeros
232 rt_entry_cache
233 rts_conn_cache
234 rts_g_t_info_ack
235 rts_max_optsize
236 rts_mod_info
237 rts_opt_arr
238 rts_opt_obj
239 rts_valid_levels_arr
240 rtsinfo
241 rtsrinit
242 rtswinit
243 sctp_asconf_default_dispatch
244 sctp_asconf_dispatch_tbl
245 sctp_conn_cache
246 sctp_conn_hash_size
247 sctp_do_reclaim
248 sctp_kmem_faddr_cache
249 sctp_kmem_ftsn_set_cache
250 sctp_kmem_set_cache
251 sctp_min_assoc_listener
252 sctp_opt_arr
253 sctp_opt_arr_size
254 sctp_pa_early_abort
255 sctp_pp_early_abort
256 sctp_propinfo_tbl
257 sctp_propinfo_count
258 sctp_recvq_tq_list_max
259 sctp_recvq_tq_task_min

```

```
260 sctp_rcvq_tq_thr_max
261 sctp_rcvq_tq_thr_min
262 sctp_sin6_null
263 sctpdebug
264 sin6_null
265 sin_null
266 skip_sctp_cksum
267 sock_dccp_downcalls
268 #endif /* ! codereview */
269 sock_rawip_downcalls
270 sock_rts_downcalls
271 sock_tcp_downcalls
272 sock_udp_downcalls
273 sqset_global_list
274 sqset_global_size
275 sqset_lock
276 squeue_cache
277 squeue_drain_ms
278 squeue_drain_ns
279 squeue_workerwait_ms
280 squeue_workerwait_tick
281 tcp_acceptor_rinit
282 tcp_acceptor_winit
283 tcp_conn_cache
284 tcp_conn_hash_size
285 tcp_do_reclaim
286 tcp_drop_ack_unsent_cnt
287 tcp_dummy_upcalls
288 tcp_early_abort
289 tcp_fallback_sock_winit
290 tcp_free_list_max_cnt
291 tcp_g_kstat
292 tcp_g_statistics
293 tcp_g_t_info_ack
294 tcp_g_t_info_ack_v6
295 tcp_icmp_source_quench
296 tcp_init_wnd_chk
297 tcp_max_init_cwnd
298 tcp_max_optsize
299 tcp_min_conn_listener
300 tcp_notsack_blk_cache
301 tcp_opt_arr
302 tcp_opt_obj
303 tcp_outbound_squeue_switch
304 tcp_propinfo_tbl
305 tcp_propinfo_count
306 tcp_random_anon_port
307 tcp_random_end_ptr
308 tcp_random_fptr
309 tcp_random_lock
310 tcp_random_rptr
311 tcp_random_state
312 tcp_randtbl
313 tcp_rinfo
314 tcp_rinitv4
315 tcp_rinitv6
316 tcp_sock_winit
317 tcp_squeue_flag
318 tcp_squeue_wput
319 tcp_static_maxpsz
320 tcp_timercache
321 tcp_tx_pull_len
322 tcp_valid_levels_arr
323 tcp_winfo
324 tcp_winit
325 tcpinfov4
```

```
326 tcpinfov6
327 tli_errs
328 tsol_strict_error
329 tun_spd_hashsize
330 udp_bind_fanout_size
331 udp_conn_cache
332 udp_fallback_sock_winit
333 udp_g_t_info_ack_ipv4
334 udp_g_t_info_ack_ipv6
335 udp_lrinit
336 udp_lwinit
337 udp_max_optsize
338 udp_mod_info
339 udp_opt_arr
340 udp_opt_obj
341 udp_propinfo_tbl
342 udp_propinfo_count
343 udp_random_anon_port
344 udp_rinitv4
345 udp_rinitv6
346 udp_valid_levels_arr
347 udp_winit
348 udpinfov4
349 udpinfov6
350 winit_arp
351 nxge_cksum_workaround
```

new/usr/src/uts/intel/ip/ip.global-objs.obj64

1

```
*****
6189 Sat Aug 18 10:37:25 2012
new/usr/src/uts/intel/ip/ip.global-objs.obj64
dccp: starting module template
*****

1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
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_addrs
46 cl_sctp_connect
47 cl_sctp_disconnect
48 cl_sctp_listen
49 cl_sctp_unlisten
50 conn_drain_nthreads
51 dccp_acceptor_rinit
52 dccp_acceptor_winit
53 dccp_bind_fanout_size
54 dccp_conn_cache
55 dccp_fallback_sock_winit
56 dccp_g_t_info_ack
57 dccp_g_t_info_ack_v6
58 dccp_max_optsize
59 dccp_opt_arr
60 dccp_opt_obj
61 dccp_propinfo_count
```

new/usr/src/uts/intel/ip/ip.global-objs.obj64

2

```
62 dccp_propinfo_tbl
63 dccp_random_anon_port
64 dccp_rinfo
65 dccp_rinitv4
66 dccp_rinitv6
67 dccp_sock_winit
68 dccp_squeue_flag
69 dccp_timercache
70 dccp_valid_levels_arr
71 dccp_winfo
72 dccp_winit
73 dccpinfov4
74 dccpinfov6
75 #endif /* ! codereview */
76 dce_cache
77 default_ip6_asp_table
78 do_tcp_fusion
79 do_tcpzcopy
80 dohwcksum
81 dummy_mod_info
82 dummymodinfo
83 dummyrmodinit
84 dummywmodinit
85 eventq_queue_in
86 eventq_queue_nic
87 eventq_queue_out
88 fsw
89 gcdb_hash
90 gcdb_hash_size
91 gcdb_lock
92 gcgrp4_hash
93 gcgrp6_hash
94 gcgrp_hash_size
95 gcgrp_lock
96 icmp_fallback_sock_winit
97 icmp_frag_size_table
98 icmp_g_t_info_ack
99 icmp_ipha
100 icmp_max_optsize
101 icmp_mod_info
102 icmp_opt_arr
103 icmp_opt_obj
104 icmp_propinfo_tbl
105 icmp_valid_levels_arr
106 icmpinfov4
107 icmpinfov6
108 icmprinitv4
109 icmprinitv6
110 icmpwinit
111 ilb_conn_cache
112 ilb_conn_cache_timeout
113 ilb_conn_hash_size
114 ilb_conn_tcp_expiry
115 ilb_conn_timer_size
116 ilb_conn_udp_expiry
117 ilb_kstat_instance
118 ilb_kmem_flags
119 ilb_nat_src_hash_size
120 ilb_nat_src_instance
121 ilb_rule_hash_size
122 ilb_sticky_cache
123 ilb_sticky_hash_size
124 ilb_sticky_expiry
125 ilb_sticky_timer_size
126 ilb_sticky_timeout
127 ill_no_arena
```

```
128 ill_null
129 inet_dev_info
130 inet_devops
131 ip6_fhtable_hash_size
132 ip6opt_ls
133 ip_cgtp_filter_rev
134 ip_conn_cache
135 ip_debug
136 ip_g_all_ones
137 ip_helper_stream_info
138 ip_helper_stream_rinit
139 ip_helper_stream_winit
140 ip_ioctl_ftbl
141 ip_loopback_mtu_v6plus
142 ip_loopback_mtuplus
143 ip_m_tbl
144 ip_max_frag_dups
145 ip_min_frag_prune_time
146 ip_minor_arena_la
147 ip_minor_arena_sa
148 ip_misc_ioctl_count
149 ip_misc_ioctl_table
150 ip_mod_info
151 ip_modclose_ackwait_ms
152 ip_ndx_ioctl_count
153 ip_ndx_ioctl_table
154 ip_poll_normal_ms
155 ip_poll_normal_ticks
156 ip_propinfo_tbl
157 ip_propinfo_count
158 ip_rput_pullups
159 ip_six_byte_all_ones
160 ip_squeue_create_callback
161 ip_squeue_enter
162 ip_squeue_fanout
163 ip_squeue_flag
164 ip_squeue_worker_wait
165 ip_thread_data
166 ip_thread_list
167 ip_thread_rwlock
168 ipcl_bind_fanout_size
169 ipcl_conn_hash_maxsize
170 ipcl_conn_hash_memfactor
171 ipcl_conn_hash_size
172 ipcl_dccp_bind_fanout_size
173 ipcl_dccp_conn_fanout_size
174 #endif /* !codereview */
175 ipcl_iptun_fanout_size
176 ipcl_raw_fanout_size
177 ipcl_udp_fanout_size
178 ipif_loopback_name
179 ipif_zero
180 ipinfov4
181 ipinfov6
182 iplrinit
183 iplwinit
184 ipmp_kstats
185 iprinitv4
186 iprinitv6
187 ipsec_action_cache
188 ipsec_hdr_pullup_needed
189 ipsec_pol_cache
190 ipsec_policy_failure_msgs
191 ipsec_sel_cache
192 ipsec_spd_hashsize
193 ipsec_weird_null_inbound_policy
```

```
194 ipv4info
195 ipv6_all_hosts_mcast
196 ipv6_all_ones
197 ipv6_all_rtrs_mcast
198 ipv6_all_v2rtrs_mcast
199 ipv6_all_zeros
200 ipv6_ll_template
201 ipv6_loopback
202 ipv6_solicited_node_mcast
203 ipv6_unspecified_group
204 ipv6info
205 ipwinit
206 ire_cache
207 ire_gw_secattr_cache
208 ire_null
209 ire_nv_arr
210 ire_nv_tbl
211 lcl_param_arr
212 mask_rnhead
213 max_keylen
214 modldrv
215 modlinkage
216 modlstrmod
217 multicast_encap_iphdr
218 nce_cache
219 ncec_cache
220 netdev_privs
221 prov_update_handle
222 radix_mask_cache
223 radix_node_cache
224 rawip_conn_cache
225 req_arr
226 rinit_arp
227 rn_mkfreelist
228 rn_ones
229 rn_zeros
230 rt_entry_cache
231 rts_conn_cache
232 rts_g_t_info_ack
233 rts_max_optsize
234 rts_mod_info
235 rts_opt_arr
236 rts_opt_obj
237 rts_valid_levels_arr
238 rtsinfo
239 rtsrinit
240 rtswinit
241 sctp_asconf_default_dispatch
242 sctp_asconf_dispatch_tbl
243 sctp_conn_cache
244 sctp_conn_hash_size
245 sctp_do_reclaim
246 sctp_kmem_faddr_cache
247 sctp_kmem_ftsn_set_cache
248 sctp_kmem_set_cache
249 sctp_min_assoc_listener
250 sctp_opt_arr
251 sctp_opt_arr_size
252 sctp_pa_early_abort
253 sctp_pp_early_abort
254 sctp_propinfo_tbl
255 sctp_propinfo_count
256 sctp_recvq_tq_list_max
257 sctp_recvq_tq_task_min
258 sctp_recvq_tq_thr_max
259 sctp_recvq_tq_thr_min
```

```
260 sctp_sin6_null
261 sctpdebug
262 sin6_null
263 sin_null
264 sock_dccp_downcalls
265 #endif /* ! codereview */
266 sock_rawip_downcalls
267 sock_rts_downcalls
268 sock_tcp_downcalls
269 sock_udp_downcalls
270 sqset_global_list
271 sqset_global_size
272 sqset_lock
273 squeue_cache
274 squeue_drain_ms
275 squeue_drain_ns
276 squeue_workerwait_ms
277 squeue_workerwait_tick
278 tcp_acceptor_rinit
279 tcp_acceptor_winit
280 tcp_conn_cache
281 tcp_conn_hash_size
282 tcp_do_reclaim
283 tcp_drop_ack_unsent_cnt
284 tcp_dummy_upcalls
285 tcp_early_abort
286 tcp_fallback_sock_winit
287 tcp_free_list_max_cnt
288 tcp_g_kstat
289 tcp_g_statistics
290 tcp_g_t_info_ack
291 tcp_g_t_info_ack_v6
292 tcp_icmp_source_quench
293 tcp_init_wnd_chk
294 tcp_max_init_cwnd
295 tcp_max_optsize
296 tcp_min_conn_listener
297 tcp_otsack_blk_cache
298 tcp_opt_arr
299 tcp_opt_obj
300 tcp_outbound_squeue_switch
301 tcp_propinfo_tbl
302 tcp_propinfo_count
303 tcp_random_anon_port
304 tcp_random_end_ptr
305 tcp_random_fptr
306 tcp_random_lock
307 tcp_random_rptr
308 tcp_random_state
309 tcp_randtbl
310 tcp_rinfo
311 tcp_rinitv4
312 tcp_rinitv6
313 tcp_sock_winit
314 tcp_squeue_flag
315 tcp_squeue_wput
316 tcp_static_maxpsz
317 tcp_timer_cache
318 tcp_tx_pull_len
319 tcp_valid_levels_arr
320 tcp_winfo
321 tcp_winit
322 tcpinfov4
323 tcpinfov6
324 tli_errs
325 tsol_strict_error
```

```
326 tun_spd_hashsize
327 udp_bind_fanout_size
328 udp_conn_cache
329 udp_fallback_sock_winit
330 udp_g_t_info_ack_ipv4
331 udp_g_t_info_ack_ipv6
332 udp_linit
333 udp_lwinit
334 udp_max_optsize
335 udp_mod_info
336 udp_opt_arr
337 udp_opt_obj
338 udp_propinfo_tbl
339 udp_propinfo_count
340 udp_random_anon_port
341 udp_rinitv4
342 udp_rinitv6
343 udp_valid_levels_arr
344 udp_winit
345 udpinfov4
346 udpinfov6
347 winit_arp
348 nxge_cksum_workaround
```