****
7942 Mon Apr 28 16:23:04 2014
new/usr/src/cmd/cmd-inet/sbin/dhcpagent/init_reboot.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
226 /*
227  * stop_init_reboot(): decides when to stop retransmitting REQUESTs
228  *
229  *   input: dhcp_smach_t *: the state machine sending the REQUESTs
230  *          unsigned int: the number of REQUESTs sent so far
231  *  output: boolean_t: B_TRUE if retransmissions should stop
232  */

234 static boolean_t
235 stop_init_reboot(dhcp_smach_t *dsmp, unsigned int n_requests)
236 {
237         if (dsmp->dsm_isv6) {
238                 uint_t nowabs, maxabs;

240                 nowabs = NSEC2MSEC(gethrtime());
241                 maxabs = NSEC2MSEC(dsmp->dsm_neg_hrtime) + DHCPV6_CNF_MAX_RD;
240                 nowabs = gethrtime() / (NANOSEC / MILLISEC);
241                 maxabs = dsmp->dsm_neg_hrtime / (NANOSEC / MILLISEC) +
242                     DHCPV6_CNF_MAX_RD;
242                 if (nowabs < maxabs) {
243                         /* Cap the timer based on the maximum */
244                         if (nowabs + dsmp->dsm_send_timeout > maxabs)
245                                 dsmp->dsm_send_timeout = maxabs - nowabs;
246                         return (B_FALSE);
247                 }
248         } else {
249                 if (n_requests < DHCP_MAX_REQUESTS)
250                         return (B_FALSE);
251         }

253         if (df_get_bool(dsmp->dsm_name, dsmp->dsm_isv6,
254             DF_VERIFIED_LEASE_ONLY)) {
255                 dhcpmsg(MSG_INFO,
256                     "unable to verify existing lease on %s; restarting",
257                     dsmp->dsm_name);
258                 dhcp_selecting(dsmp);
259                 return (B_TRUE);
260         }

262         if (dsmp->dsm_isv6) {
263                 dhcpmsg(MSG_INFO, "no Reply to Confirm, using remainder of "
264                     "existing lease on %s", dsmp->dsm_name);
265         } else {
266                 dhcpmsg(MSG_INFO, "no ACK/NAK to INIT_REBOOT REQUEST, "
267                     "using remainder of existing lease on %s", dsmp->dsm_name);
268         }

270         /*
271          * We already stuck our old ack in dsmp->dsm_ack and relativized the
272          * packet times, so we can just pretend that the server sent it to us
273          * and move to bound.  If that fails, fall back to selecting.
274          */

276         if (dhcp_bound(dsmp, NULL)) {
277                 if (dsmp->dsm_isv6) {
278                         if (!save_server_id(dsmp, dsmp->dsm_ack))
279                                 goto failure;
280                         server_unicast_option(dsmp, dsmp->dsm_ack);
281                 }
```



```
282         } else {
283 failure:
284                 dhcpmsg(MSG_INFO, "unable to use saved lease on %s; restarting",
285                     dsmp->dsm_name);
286                 dhcp_selecting(dsmp);
287         }

289         return (B_TRUE);
290 }
```

_____***unchanged_portion_omitted_***

****
82094 Mon Apr 28 16:23:05 2014
new/usr/src/cmd/cmd-inet/usr.lib/in.mpathd/mpd_probe.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
2726 static int
2727 ns2ms(int64_t ns)
2728 {
2729         return (NSEC2MSEC(ns));
2729         return (ns / (NANOSEC / MILLISEC));
2730 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35924 Mon Apr 28 16:23:05 2014
new/usr/src/cmd/cmd-inet/usr.sbin/ipmpstat/ipmpstat.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
______unchanged_portion_omitted_

 155 #define IPMPSTAT_NCOL   80
 156 #define NS2FLOATMS(ns)  (NSEC2MSEC((float)(ns)))
 156 #define NS2FLOATMS(ns)  ((float)(ns) / (NANOSEC / MILLISEC))
 157 #define MS2FLOATSEC(ms) ((float)(ms) / 1000)

 159 static const char       *progname;
 160 static hrtime_t         probe_output_start;
 161 static ipmpstat_opt_t   opt;
 162 static ofmt_handle_t    ofmt;
 163 static ipmpstat_enum_t  addr_state[], group_state[], if_state[], if_link[];
 164 static ipmpstat_enum_t  if_probe[], targ_mode[];
 165 static ofmt_field_t     addr_fields[], group_fields[], if_fields[];
 166 static ofmt_field_t     probe_fields[], targ_fields[];
 167 static ipmpstat_cbfunc_t walk_addr_cbfunc, walk_if_cbfunc;
 168 static ipmpstat_cbfunc_t info_output_cbfunc, targinfo_output_cbfunc;
 169 static ipmpstat_walker_t walk_addr, walk_if, walk_group;

 171 static int probe_event(sysevent_t *, void *);
 172 static void probe_output(ipmp_handle_t, ofmt_handle_t);
 173 static void ofmt_output(ofmt_handle_t, ipmp_handle_t, void *);
 174 static void enum2str(const ipmpstat_enum_t *, int, char *, uint_t);
 175 static void sockaddr2str(const struct sockaddr_storage *, char *, uint_t);
 176 static void sighandler(int);
 177 static void usage(void);
 178 static void die(const char *, ...);
 179 static void die_ipmperr(int, const char *, ...);
 180 static void warn(const char *, ...);
 181 static void warn_ipmperr(int, const char *, ...);

 183 int
 184 main(int argc, char **argv)
 185 {
 186         int c;
 187         int err;
 188         const char *ofields = NULL;
 189         ofmt_status_t ofmterr;
 190         ofmt_field_t *fields = NULL;
 191         uint_t ofmtflags = 0;
 192         ipmp_handle_t ih;
 193         ipmp_qcontext_t qcontext = IPMP_QCONTEXT_SNAP;
 194         ipmpstat_cbfunc_t *cbfunc;
 195         ipmpstat_walker_t *walker;
 196         char errbuf[OFMT_BUFSIZE];

 198         if ((progname = strrchr(argv[0], '/')) == NULL)
 199                 progname = argv[0];
 200         else
 201                 progname++;

 203         (void) setlocale(LC_ALL, "");
 204         (void) textdomain(TEXT_DOMAIN);

 206         while ((c = getopt(argc, argv, "nLPo:agipt")) != EOF) {
 207                 if (fields != NULL && strchr("agipt", c) != NULL)
 208                         die("only one output format may be specified\n");

 210                 switch (c) {
 211                 case 'n':
 212                         opt |= IPMPSTAT_OPT_NUMERIC;
```



```
 213                         break;
 214                 case 'L':
 215                         /* Undocumented option: for testing use ONLY */
 216                         qcontext = IPMP_QCONTEXT_LIVE;
 217                         break;
 218                 case 'P':
 219                         opt |= IPMPSTAT_OPT_PARSABLE;
 220                         ofmtflags |= OFMT_PARSABLE;
 221                         break;
 222                 case 'o':
 223                         ofields = optarg;
 224                         break;
 225                 case 'a':
 226                         walker = walk_addr;
 227                         cbfunc = info_output_cbfunc;
 228                         fields = addr_fields;
 229                         break;
 230                 case 'g':
 231                         walker = walk_group;
 232                         cbfunc = info_output_cbfunc;
 233                         fields = group_fields;
 234                         break;
 235                 case 'i':
 236                         walker = walk_if;
 237                         cbfunc = info_output_cbfunc;
 238                         fields = if_fields;
 239                         break;
 240                 case 'p':
 241                         fields = probe_fields;
 242                         break;
 243                 case 't':
 244                         walker = walk_if;
 245                         cbfunc = targinfo_output_cbfunc;
 246                         fields = targ_fields;
 247                         break;
 248                 default:
 249                         usage();
 250                         break;
 251                 }
 252         }

 254         if (argc > optind || fields == NULL)
 255                 usage();

 257         /*
 258          * Open a handle to the formatted output engine.
 259          */
 260         ofmterr = ofmt_open(ofields, fields, ofmtflags, IPMPSTAT_NCOL, &ofmt);
 261         if (ofmterr != OFMT_SUCCESS) {
 262                 /*
 263                  * If some fields were badly formed in human-friendly mode, we
 264                  * emit a warning and continue.  Otherwise exit immediately.
 265                  */
 266                 (void) ofmt_strerror(ofmt, ofmterr, errbuf, sizeof (errbuf));
 267                 if (ofmterr != OFMT_EBADFIELDS || (opt & IPMPSTAT_OPT_PARSABLE))
 268                         die("%s\n", errbuf);
 269                 else
 270                         warn("%s\n", errbuf);
 271         }

 273         /*
 274          * Obtain the window size and monitor changes to the size.  This data
 275          * is used to redisplay the output headers when necessary.
 276          */
 277         (void) sigset(SIGWINCH, sighandler);
```

```
279         if ((err = ipmp_open(&ih)) != IPMP_SUCCESS)
280                 die_ipmperr(err, "cannot create IPMP handle");

282         if (ipmp_ping_daemon(ih) != IPMP_SUCCESS)
283                 die("cannot contact in.mpathd(1M) -- is IPMP in use?\n");

285         /*
286          * If we've been asked to display probes, then call the probe output
287          * function.  Otherwise, snapshot IPMP state (or use live state) and
288          * invoke the specified walker with the specified callback function.
289          */
290         if (fields == probe_fields) {
291                 probe_output(ih, ofmt);
292         } else {
293                 if ((err = ipmp_setqcontext(ih, qcontext)) != IPMP_SUCCESS) {
294                         if (qcontext == IPMP_QCONTEXT_SNAP)
295                                 die_ipmperr(err, "cannot snapshot IPMP state");
296                         else
297                                 die_ipmperr(err, "cannot use live IPMP state");
298                 }
299                 (*walker)(ih, cbfunc, ofmt);
300         }

302         ofmt_close(ofmt);
303         ipmp_close(ih);

305         return (EXIT_SUCCESS);
306 }
```

_____***unchanged_portion_omitted_***

****
130930 Mon Apr 28 16:23:05 2014
new/usr/src/cmd/idmap/idmapd/dbutils.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
 329 /*
 330  * This is the SQLite database busy handler that retries the SQL
 331  * operation until it is successful.
 332  */
 333 int
 334 /* LINTED E_FUNC_ARG_UNUSED */
 335 idmap_sqlite_busy_handler(void *arg, const char *table_name, int count)
 336 {
 337         struct idmap_busy       *busy = arg;
 338         int                     delay;
 339         struct timespec         rqtp;

 341         if (count == 1)  {
 342                 busy->total = 0;
 343                 busy->sec = 2;
 344         }
 345         if (busy->total > 1000 * busy->sec) {
 346                 idmapdlog(LOG_DEBUG,
 347                     "Thread %d waited %d sec for the %s database",
 348                     pthread_self(), busy->sec, busy->name);
 349                 busy->sec++;
 350         }

 352         if (count <= busy->delay_size) {
 353                 delay = busy->delays[count-1];
 354         } else {
 355                 delay = busy->delays[busy->delay_size - 1];
 356         }
 357         busy->total += delay;
 358         rqtp.tv_sec = 0;
 359         rqtp.tv_nsec = MSEC2NSEC(delay);
 359         rqtp.tv_nsec = delay * (NANOSEC / MILLISEC);
 360         (void) nanosleep(&rqtp, NULL);
 361         return (1);
 362 }
```

_____***unchanged_portion_omitted_***

****
12715 Mon Apr 28 16:23:05 2014
new/usr/src/cmd/mdb/common/modules/svc.configd/configd.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ctf.h>

#include <configd.h>

mdb_ctf_id_t request_enum;
mdb_ctf_id_t response_enum;
mdb_ctf_id_t ptr_type_enum;
mdb_ctf_id_t thread_state_enum;

hrtime_t max_time_seen;

static void
enum_lookup(char *out, size_t size, mdb_ctf_id_t id, int val,
    const char *prefix, const char *prefix2)
{
        const char *cp;
        size_t len = strlen(prefix);
        size_t len2 = strlen(prefix2);

        if ((cp = mdb_ctf_enum_name(id, val)) != NULL) {
                if (strncmp(cp, prefix, len) == 0)
                        cp += len;
                if (strncmp(cp, prefix2, len2) == 0)
                        cp += len2;
                (void) strlcpy(out, cp, size);
        } else {
                mdb_snprintf(out, size, "? (%d)", val);
        }
}
```

*_____unchanged_portion_omitted_*

```
static int
```



```
request_log(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        request_log_entry_t cur;
        hrtime_t dur;
        hrtime_t dursec;
        hrtime_t durnsec;
        char durstr[20];
        char stampstr[20];
        char requstr[30];
        char respstr[30];
        char typestr[30];
        uintptr_t node = 0;
        uintptr_t client = 0;
        uint64_t clientid = 0;

        int idx;
        int opt_v = FALSE;                      /* verbose */

        if (!(flags & DCMD_ADDRSPEC)) {
                if (mdb_walk_dcmd("configd_log", "configd_log", argc,
                    argv) == -1) {
                        mdb_warn("can't walk 'configd_log'");
                        return (DCMD_ERR);
                }
                return (DCMD_OK);
        }

        if (mdb_getopts(argc, argv,
            'c', MDB_OPT_UINTPTR, &client,
            'i', MDB_OPT_UINT64, &clientid,
            'n', MDB_OPT_UINTPTR, &node,
            'v', MDB_OPT_SETBITS, TRUE, &opt_v, NULL) != argc)
                return (DCMD_USAGE);

        if (DCMD_HDRSPEC(flags)) {
                mdb_printf("%<u>%-?s %-4s %-14s %9s %-22s %-17s\n%</u>",
                    "ADDR", "THRD", "START", "DURATION", "REQUEST",
                    "RESPONSE");
        }

        if (mdb_vread(&cur, sizeof (cur), addr) == -1) {
                mdb_warn("couldn't read log entry at %p", addr);
                return (DCMD_ERR);
        }

        /*
         * apply filters, if any.
         */
        if (clientid != 0 && clientid != cur.rl_clientid)
                return (DCMD_OK);

        if (client != 0 && client != (uintptr_t)cur.rl_client)
                return (DCMD_OK);

        if (node != 0) {
                for (idx = 0; idx < MIN(MAX_PTRS, cur.rl_num_ptrs); idx++) {
                        if ((uintptr_t)cur.rl_ptrs[idx].rlp_data == node) {
                                node = 0;               /* found it */
                                break;
                        }
                }
                if (node != 0)
                        return (DCMD_OK);
        }

        enum_lookup(requstr, sizeof (requstr), request_enum, cur.rl_request,
```

```
242             "REP_PROTOCOL_", "");

244         if (cur.rl_end != 0) {
245                 enum_lookup(respstr, sizeof (respstr), response_enum,
246                     cur.rl_response, "REP_PROTOCOL_", "FAIL_");

248                 dur = cur.rl_end - cur.rl_start;
249                 dursec = dur / NANOSEC;
250                 durnsec = dur % NANOSEC;

252                 if (dursec <= 9)
253                         mdb_snprintf(durstr, sizeof (durstr),
254                             "%lld.%06lld",
255                             dursec, durnsec / (NANOSEC / MICROSEC));
256                 else if (dursec <= 9999)
257                         mdb_snprintf(durstr, sizeof (durstr),
258                             "%lld.%03lld",
259                             dursec, NSEC2MSEC(durnsec));
261                             dursec, durnsec / (NANOSEC / MILLISEC));
260                 else
261                         mdb_snprintf(durstr, sizeof (durstr),
262                             "%lld", dursec);
263         } else {
264                 (void) strcpy(durstr, "-");
265                 (void) strcpy(respstr, "-");
266         }

268         if (max_time_seen != 0 && max_time_seen >= cur.rl_start) {
269                 dur = max_time_seen - cur.rl_start;
270                 dursec = dur / NANOSEC;
271                 durnsec = dur % NANOSEC;

273                 if (dursec <= 99ULL)
274                         mdb_snprintf(stampstr, sizeof (stampstr),
275                             "-%lld.%09lld", dursec, durnsec);
276                 else if (dursec <= 99999ULL)
277                         mdb_snprintf(stampstr, sizeof (stampstr),
278                             "-%lld.%06lld",
279                             dursec, durnsec / (NANOSEC / MICROSEC));
280                 else if (dursec <= 99999999ULL)
281                         mdb_snprintf(stampstr, sizeof (stampstr),
282                             "-%lld.%03lld",
283                             dursec, NSEC2MSEC(durnsec));
285                             dursec, durnsec / (NANOSEC / MILLISEC));
284                 else
285                         mdb_snprintf(stampstr, sizeof (stampstr),
286                             "-%lld", dursec);
287         } else {
288                 (void) strcpy(stampstr, "-");
289         }

291         mdb_printf("%0?x %4d T%13s %9s %-22s %-17s\n",
292             addr, cur.rl_tid, stampstr, durstr, requstr, respstr);

294         if (opt_v) {
295                 mdb_printf("\tclient: %?p (%d)\tptrs: %d\tstamp: %llx\n",
296                     cur.rl_client, cur.rl_clientid, cur.rl_num_ptrs,
297                     cur.rl_start);
298                 for (idx = 0; idx < MIN(MAX_PTRS, cur.rl_num_ptrs); idx++) {
299                         enum_lookup(typestr, sizeof (typestr), ptr_type_enum,
300                             cur.rl_ptrs[idx].rlp_type, "RC_PTR_TYPE_", "");
301                         mdb_printf("\t\t%-7s %5d %?p %?p\n", typestr,
302                             cur.rl_ptrs[idx].rlp_id, cur.rl_ptrs[idx].rlp_ptr,
303                             cur.rl_ptrs[idx].rlp_data);
304                 }
305                 mdb_printf("\n");
```



```
306         }
307         return (DCMD_OK);
308 }
```

_____***unchanged_portion_omitted_***

****
44261 Mon Apr 28 16:23:06 2014
new/usr/src/cmd/rcap/rcapd/rcapd_main.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
1023 /*
1024  * Print, for debugging purposes, each collection's interval statistics.
1025  */
1026 /*ARGSUSED*/
1027 static int
1028 simple_report_collection_cb(lcollection_t *lcol, void *arg)
1029 {
1030 #define DELTA(field) \
1031         (unsigned long long)( \
1032             (lcol->lcol_stat.field - lcol->lcol_stat_old.field))

1034         debug("%s %s status: succeeded/attempted (k): %llu/%llu, "
1035             "ineffective/scans/unenforced/samplings:  %llu/%llu/%llu/%llu, RSS "
1036             "min/max (k): %llu/%llu, cap %llu kB, processes/thpt: %llu/%llu, "
1037             "%llu scans over %llu ms\n",
1038             (lcol->lcol_id.rcid_type == RCIDT_PROJECT ? "project" : "zone"),
1039             lcol->lcol_name,
1040             DELTA(lcols_pg_eff), DELTA(lcols_pg_att),
1041             DELTA(lcols_scan_ineffective), DELTA(lcols_scan),
1042             DELTA(lcols_unenforced_cap), DELTA(lcols_rss_sample),
1043             (unsigned long long)lcol->lcol_stat.lcols_min_rss,
1044             (unsigned long long)lcol->lcol_stat.lcols_max_rss,
1045             (unsigned long long)lcol->lcol_rss_cap,
1046             (unsigned long long)(lcol->lcol_stat.lcols_proc_in -
1047             lcol->lcol_stat.lcols_proc_out), DELTA(lcols_proc_out),
1048             DELTA(lcols_scan_count),
1049             NSEC2MSEC(DELTA(lcols_scan_time_complete)));
1048             DELTA(lcols_scan_count), DELTA(lcols_scan_time_complete) / (NANOSEC
1049             / MILLISEC));

1051 #undef DELTA

1053         return (0);
1054 }
```

_____***unchanged_portion_omitted_***

****
10980 Mon Apr 28 16:23:06 2014
new/usr/src/cmd/rcap/rcapstat/rcapstat.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
254 /*
255  * Print each collection's interval statistics.
256  */
257 /*ARGSUSED*/
258 static void
259 print_unformatted_stats(void)
260 {
261         col_t *col;

263 #define DELTA(field) \
264         (col->col_src_stat.field - col->col_old_stat.field)

266         col = col_head;
267         while (col != NULL) {
268                 if (bcmp(&col->col_src_stat, &col->col_old_stat,
269                     sizeof (col->col_src_stat)) == 0) {
270                         col = col->col_next;
271                         continue;
272                 }
273                 (void) printf("%s %s status: succeeded/attempted (k): "
274                     "%llu/%llu, ineffective/scans/unenforced/samplings:  "
275                     "%llu/%llu/%llu/%llu, RSS min/max (k): %llu/%llu, cap %llu "
276                     "kB, processes/thpt: %llu/%llu, %llu scans over %lld ms\n",
277                     mode, col->col_name, DELTA(lcols_pg_eff),
278                     DELTA(lcols_pg_att), DELTA(lcols_scan_ineffective),
279                     DELTA(lcols_scan), DELTA(lcols_unenforced_cap),
280                     DELTA(lcols_rss_sample), col->col_src_stat.lcols_min_rss,
281                     col->col_src_stat.lcols_max_rss, col->col_rsslimit,
282                     (col->col_src_stat.lcols_proc_in -
283                     col->col_old_stat.lcols_proc_out), DELTA(lcols_proc_out),
284                     DELTA(lcols_scan_count),
285                     NSEC2MSEC(DELTA(lcols_scan_time_complete)));
284                     DELTA(lcols_scan_count), DELTA(lcols_scan_time_complete) /
285                     (NANOSEC / MILLISEC));
286                 col->col_old_stat = col->col_src_stat;

288                 col = col->col_next;
289         }

291         if (global)
292                 (void) printf(gettext("physical memory utilization: %3u%%   "
293                     "cap enforcement threshold: %3u%%\n"), hdr.rs_pressure_cur,
294                     hdr.rs_pressure_cap);
295 #undef DELTA
296 }
```

_____***unchanged_portion_omitted_***

****
58084 Mon Apr 28 16:23:06 2014
new/usr/src/cmd/zlogin/zlogin.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
 594 /*
 595  * process_user_input watches the input stream for the escape sequence for
 596  * 'quit' (by default, tilde-period).  Because we might be fed just one
 597  * keystroke at a time, state associated with the user input (are we at the
 598  * beginning of the line?  are we locally echoing the next character?) is
 599  * maintained by beginning_of_line and local_echo across calls to the routine.
 600  * If the write to outfd fails, we'll try to read from infd in an attempt
 601  * to prevent deadlock between the two processes.
 602  *
 603  * This routine returns -1 when the 'quit' escape sequence has been issued,
 604  * or an error is encountered, 1 if stdin is EOF, and 0 otherwise.
 605  */
 606 static int
 607 process_user_input(int outfd, int infd)
 608 {
 609         static boolean_t beginning_of_line = B_TRUE;
 610         static boolean_t local_echo = B_FALSE;
 611         char ibuf[ZLOGIN_BUFSIZ];
 612         int nbytes;
 613         char *buf = ibuf;
 614         char c = *buf;

 616         nbytes = read(STDIN_FILENO, ibuf, ZLOGIN_RDBUFSIZ);
 617         if (nbytes == -1 && (errno != EINTR || dead))
 618                 return (-1);

 620         if (nbytes == -1)       /* The read was interrupted. */
 621                 return (0);

 623         /* 0 read means EOF, close the pipe to the child */
 624         if (nbytes == 0)
 625                 return (1);

 627         for (c = *buf; nbytes > 0; c = *buf, --nbytes) {
 628                 buf++;
 629                 if (beginning_of_line && !nocmdchar) {
 630                         beginning_of_line = B_FALSE;
 631                         if (c == cmdchar) {
 632                                 local_echo = B_TRUE;
 633                                 continue;
 634                         }
 635                 } else if (local_echo) {
 636                         local_echo = B_FALSE;
 637                         if (c == '.' || c == effective_termios.c_cc[VEOF]) {
 638                                 char cc[CANONIFY_LEN];

 640                                 canonify(c, cc);
 641                                 (void) write(STDOUT_FILENO, &cmdchar, 1);
 642                                 (void) write(STDOUT_FILENO, cc, strlen(cc));
 643                                 return (-1);
 644                         }
 645                 }
 646 retry:
 647                 if (write(outfd, &c, 1) <= 0) {
 648                         /*
 649                          * Since the fd we are writing to is opened with
 650                          * O_NONBLOCK it is possible to get EAGAIN if the
 651                          * pipe is full.  One way this could happen is if we
 652                          * are writing a lot of data into the pipe in this loop
```



```
 653                          * and the application on the other end is echoing that
 654                          * data back out to its stdout.  The output pipe can
 655                          * fill up since we are stuck here in this loop and not
 656                          * draining the other pipe.  We can try to read some of
 657                          * the data to see if we can drain the pipe so that the
 658                          * application can continue to make progress.  The read
 659                          * is non-blocking so we won't hang here.  We also wait
 660                          * a bit before retrying since there could be other
 661                          * reasons why the pipe is full and we don't want to
 662                          * continuously retry.
 663                          */
 664                         if (errno == EAGAIN) {
 665                                 struct timespec rqtp;
 666                                 int ln;
 667                                 char obuf[ZLOGIN_BUFSIZ];

 669                                 if ((ln = read(infd, obuf, ZLOGIN_BUFSIZ)) > 0)
 670                                         (void) write(STDOUT_FILENO, obuf, ln);

 672                                 /* sleep for 10 milliseconds */
 673                                 rqtp.tv_sec = 0;
 674                                 rqtp.tv_nsec = MSEC2NSEC(10);
 674                                 rqtp.tv_nsec = 10 * (NANOSEC / MILLISEC);
 675                                 (void) nanosleep(&rqtp, NULL);
 676                                 if (!dead)
 677                                         goto retry;
 678                         }

 680                         return (-1);
 681                 }
 682                 beginning_of_line = (c == '\r' || c == '\n' ||
 683                     c == effective_termios.c_cc[VKILL] ||
 684                     c == effective_termios.c_cc[VEOL] ||
 685                     c == effective_termios.c_cc[VSUSP] ||
 686                     c == effective_termios.c_cc[VINTR]);
 687         }
 688         return (0);
 689 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28015 Mon Apr 28 16:23:06 2014
new/usr/src/lib/libdhcpagent/common/dhcpagent_ipc.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
______unchanged_portion_omitted_

 961 /*
 962  * dhcp_ipc_timed_read(): reads from a descriptor using a maximum timeout
 963  *
 964  *   input: int: the file descriptor to read from
 965  *          void *: the buffer to read into
 966  *          unsigned int: the total length of data to read
 967  *          int *: the number of milliseconds to wait; the number of
 968  *                 milliseconds left are returned (-1 is "forever")
 969  *  output: int: DHCP_IPC_SUCCESS on success, DHCP_IPC_E_* otherwise
 970  */

 972 static int
 973 dhcp_ipc_timed_read(int fd, void *buffer, unsigned int length, int *msec)
 974 {
 975         unsigned int    n_total = 0;
 976         ssize_t         n_read;
 977         struct pollfd   pollfd;
 978         hrtime_t        start, end;
 979         int             retv;

 981         pollfd.fd       = fd;
 982         pollfd.events   = POLLIN;

 984         while (n_total < length) {

 986                 start = gethrtime();

 988                 retv = poll(&pollfd, 1, *msec);
 989                 if (retv == 0) {
 990                         /* This can happen only if *msec is not -1 */
 991                         *msec = 0;
 992                         return (DHCP_IPC_E_TIMEOUT);
 993                 }

 995                 if (*msec != -1) {
 996                         end = gethrtime();
 997                         *msec -= NSEC2MSEC(end - start);
 997                         *msec -= (end - start) / (NANOSEC / MILLISEC);
 998                         if (*msec < 0)
 999                                 *msec = 0;
1000                 }

1002                 if (retv == -1) {
1003                         if (errno != EINTR)
1004                                 return (DHCP_IPC_E_POLL);
1005                         else if (*msec == 0)
1006                                 return (DHCP_IPC_E_TIMEOUT);
1007                         continue;
1008                 }

1010                 if (!(pollfd.revents & POLLIN)) {
1011                         errno = EINVAL;
1012                         return (DHCP_IPC_E_POLL);
1013                 }

1015                 n_read = read(fd, (caddr_t)buffer + n_total, length - n_total);

1017                 if (n_read == -1) {
1018                         if (errno != EINTR)
```



```
1019                                 return (DHCP_IPC_E_READ);
1020                         else if (*msec == 0)
1021                                 return (DHCP_IPC_E_TIMEOUT);
1022                         continue;
1023                 }

1025                 if (n_read == 0) {
1026                         return (n_total == 0 ? DHCP_IPC_E_EOF :
1027                             DHCP_IPC_E_PROTO);
1028                 }

1030                 n_total += n_read;

1032                 if (*msec == 0 && n_total < length)
1033                         return (DHCP_IPC_E_TIMEOUT);
1034         }

1036         return (DHCP_IPC_SUCCESS);
1037 }
______unchanged_portion_omitted_
```

```
**********************************************************
   50920 Mon Apr 28 16:23:06 2014
new/usr/src/lib/libdlpi/common/libdlpi.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
______unchanged_portion_omitted_

1282 /*
1283  * Get a DLPI control message and/or data message from a stream. The inputs
1284  * for this function are:
1285  *      dlpi_impl_t *dip:       internal dlpi handle
1286  *      int msec:               timeout to wait for message
1287  *      dlpi_msg_t *dlreplyp:   reply message structure, the message size
1288  *                              member on return stores actual size received
1289  *      t_uscalar_t dlreqprim:  requested primitive
1290  *      t_uscalar_t dlreplyprim:acknowledged primitive in response to request
1291  *      size_t dlreplyminsz:    minimum size of acknowledged primitive size
1292  *      void *databuf:          data buffer
1293  *      size_t *datalenp:       data buffer len
1294  *      size_t *totdatalenp:    total data received. Greater than 'datalenp' if
1295  *                              actual data received is larger than 'databuf'
1296  * Function returns DLPI_SUCCESS if requested message is retrieved
1297  * otherwise returns error code or timeouts. If a notification arrives on
1298  * the stream the callback is notified. However, error returned during the
1299  * handling of notification is ignored as it would be confusing to actual caller
1300  * of this function.
1301  */
1302 static int
1303 i_dlpi_strgetmsg(dlpi_impl_t *dip, int msec, dlpi_msg_t *dlreplyp,
1304     t_uscalar_t dlreqprim, t_uscalar_t dlreplyprim, size_t dlreplyminsz,
1305     void *databuf, size_t *datalenp, size_t *totdatalenp)
1306 {
1307         int                     retval;
1308         int                     flags;
1309         int                     fd = dip->dli_fd;
1310         struct strbuf           ctl, data;
1311         struct pollfd           pfd;
1312         hrtime_t                start, current;
1313         long                    bufc[DLPI_CHUNKSIZE / sizeof (long)];
1314         long                    bufd[DLPI_CHUNKSIZE / sizeof (long)];
1315         union DL_primitives     *dlprim;
1316         dl_notify_ind_t         *dlnotif;
1317         boolean_t               infinite = (msec < 0);  /* infinite timeout */

1319         /*
1320          * dlreplyp and databuf can be NULL at the same time, to force a check
1321          * for pending events on the DLPI link instance; dlpi_enabnotify(3DLPI).
1322          * this will be true more so for DLPI_RAW mode with notifications
1323          * enabled.
1324          */
1325         if ((databuf == NULL && datalenp != NULL) ||
1326             (databuf != NULL && datalenp == NULL))
1327                 return (DLPI_EINVAL);

1329         pfd.fd = fd;
1330         pfd.events = POLLIN | POLLPRI;

1332         ctl.buf = (dlreplyp == NULL) ? bufc : (void *)dlreplyp->dlm_msg;
1333         ctl.len = 0;
1334         ctl.maxlen = (dlreplyp == NULL) ? sizeof (bufc) : dlreplyp->dlm_msgsz;

1336         data.buf = (databuf == NULL) ? bufd : databuf;
1337         data.len = 0;
1338         data.maxlen = (databuf == NULL) ? sizeof (bufd): *datalenp;

1340         for (;;) {
```



```
1341                 if (!infinite)
1342                         start = NSEC2MSEC(gethrtime());
1342                         start = gethrtime() / (NANOSEC / MILLISEC);

1344                 switch (poll(&pfd, 1, msec)) {
1345                 default:
1346                         if (pfd.revents & POLLHUP)
1347                                 return (DL_SYSERR);
1348                         break;
1349                 case 0:
1350                         return (DLPI_ETIMEDOUT);
1351                 case -1:
1352                         return (DL_SYSERR);
1353                 }

1355                 flags = 0;
1356                 if ((retval = getmsg(fd, &ctl, &data, &flags)) < 0)
1357                         return (DL_SYSERR);

1359                 if (totdatalenp != NULL)
1360                         *totdatalenp = data.len;

1362                 /*
1363                  * The supplied DLPI_CHUNKSIZE sized buffers are large enough
1364                  * to retrieve all valid DLPI responses in one iteration.
1365                  * If MORECTL or MOREDATA is set, we are not interested in the
1366                  * remainder of the message. Temporary buffers are used to
1367                  * drain the remainder of this message.
1368                  * The special case we have to account for is if
1369                  * a higher priority messages is enqueued  whilst handling
1370                  * this condition. We use a change in the flags parameter
1371                  * returned by getmsg() to indicate the message has changed.
1372                  */
1373                 while (retval & (MORECTL | MOREDATA)) {
1374                         struct strbuf   cscratch, dscratch;
1375                         int             oflags = flags;

1377                         cscratch.buf = (char *)bufc;
1378                         dscratch.buf = (char *)bufd;
1379                         cscratch.len = dscratch.len = 0;
1380                         cscratch.maxlen = dscratch.maxlen =
1381                             sizeof (bufc);

1383                         if ((retval = getmsg(fd, &cscratch, &dscratch,
1384                             &flags)) < 0)
1385                                 return (DL_SYSERR);

1387                         if (totdatalenp != NULL)
1388                                 *totdatalenp += dscratch.len;
1389                         /*
1390                          * In the special case of higher priority
1391                          * message received, the low priority message
1392                          * received earlier is discarded, if no data
1393                          * or control message is left.
1394                          */
1395                         if ((flags != oflags) &&
1396                             !(retval & (MORECTL | MOREDATA)) &&
1397                             (cscratch.len != 0)) {
1398                                 ctl.len = MIN(cscratch.len, DLPI_CHUNKSIZE);
1399                                 if (dlreplyp != NULL)
1400                                         (void) memcpy(dlreplyp->dlm_msg, bufc,
1401                                             ctl.len);
1402                                 break;
1403                         }
1404                 }
```

```
1406                /*
1407                 * Check if DL_NOTIFY_IND message received. If there is one,
1408                 * notify the callback function(s) and continue processing the
1409                 * requested message.
1410                 */
1411                if (dip->dli_notifylistp != NULL &&
1412                    ctl.len >= (int)(sizeof (t_uscalar_t)) &&
1413                    *(t_uscalar_t *)(void *)ctl.buf == DL_NOTIFY_IND) {
1414                        /* process properly-formed DL_NOTIFY_IND messages */
1415                        if (ctl.len >= DL_NOTIFY_IND_SIZE) {
1416                                dlnotif = (dl_notify_ind_t *)(void *)ctl.buf;
1417                                (void) i_dlpi_notifyind_process(dip, dlnotif);
1418                        }
1419                        goto update_timer;
1420                }

1422                /*
1423                 * If we were expecting a data message, and we got one, set
1424                 * *datalenp.  If we aren't waiting on a control message, then
1425                 * we're done.
1426                 */
1427                if (databuf != NULL && data.len >= 0) {
1428                        *datalenp = data.len;
1429                        if (dlreplyp == NULL)
1430                                break;
1431                }

1433                /*
1434                 * If we were expecting a control message, and the message
1435                 * we received is at least big enough to be a DLPI message,
1436                 * then verify it's a reply to something we sent.  If it
1437                 * is a reply to something we sent, also verify its size.
1438                 */
1439                if (dlreplyp != NULL && ctl.len >= sizeof (t_uscalar_t)) {
1440                        dlprim = dlreplyp->dlm_msg;
1441                        if (dlprim->dl_primitive == dlreplyprim) {
1442                                if (ctl.len < dlreplyminsz)
1443                                        return (DLPI_EBADMSG);
1444                                dlreplyp->dlm_msgsz = ctl.len;
1445                                break;
1446                        } else if (dlprim->dl_primitive == DL_ERROR_ACK) {
1447                                if (ctl.len < DL_ERROR_ACK_SIZE)
1448                                        return (DLPI_EBADMSG);

1450                                /* Is it ours? */
1451                                if (dlprim->error_ack.dl_error_primitive ==
1452                                    dlreqprim)
1453                                        break;
1454                        }
1455                }
1456 update_timer:
1457                if (!infinite) {
1458                        current = NSEC2MSEC(gethrtime());
1458                        current = gethrtime() / (NANOSEC / MILLISEC);
1459                        msec -= (current - start);

1461                        if (msec <= 0)
1462                                return (DLPI_ETIMEDOUT);
1463                }
1464        }

1466        return (DLPI_SUCCESS);
1467 }
```

_____*unchanged_portion_omitted_*

****
11383 Mon Apr 28 16:23:07 2014
new/usr/src/lib/libinetutil/common/tq.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
 75 /*
 76  * insert_timer(): inserts a timer node into a tq's timer list
 77  *
 78  *   input: iu_tq_t *: the timer queue
 79  *          iu_timer_node_t *: the timer node to insert into the list
 80  *          uint64_t: the number of milliseconds before this timer fires
 81  *  output: void
 82  */

 84 static void
 85 insert_timer(iu_tq_t *tq, iu_timer_node_t *node, uint64_t msec)
 86 {
 87         iu_timer_node_t *after = NULL;

 89         /*
 90          * find the node to insert this new node "after".  we do this
 91          * instead of the more intuitive "insert before" because with
 92          * the insert before approach, a null 'before' node pointer
 93          * is overloaded in meaning (it could be null because there
 94          * are no items in the list, or it could be null because this
 95          * is the last item on the list, which are very different cases).
 96          */

 98         node->iutn_abs_timeout = gethrtime() + MSEC2NSEC(msec);
 98         node->iutn_abs_timeout = gethrtime() + (msec * (NANOSEC / MILLISEC));

100         if (tq->iutq_head != NULL &&
101             tq->iutq_head->iutn_abs_timeout < node->iutn_abs_timeout)
102                 for (after = tq->iutq_head; after->iutn_next != NULL;
103                     after = after->iutn_next)
104                         if (after->iutn_next->iutn_abs_timeout >
105                             node->iutn_abs_timeout)
106                                 break;

108         node->iutn_next = after ? after->iutn_next : tq->iutq_head;
109         node->iutn_prev = after;
110         if (after == NULL)
111                 tq->iutq_head = node;
112         else
113                 after->iutn_next = node;

115         if (node->iutn_next != NULL)
116                 node->iutn_next->iutn_prev = node;
117 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   47383 Mon Apr 28 16:23:07 2014
new/usr/src/lib/libldap5/sources/ldap/common/os-ip.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
```

```
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * The contents of this file are subject to the Netscape Public
 * License Version 1.1 (the "License"); you may not use this file
 * except in compliance with the License. You may obtain a copy of
 * the License at http://www.mozilla.org/NPL/
 *
 * Software distributed under the License is distributed on an "AS
 * IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or
 * implied. See the License for the specific language governing
 * rights and limitations under the License.
 *
 * The Original Code is Mozilla Communicator client code, released
 * March 31, 1998.
 *
 * The Initial Developer of the Original Code is Netscape
 * Communications Corporation. Portions created by Netscape are
 * Copyright (C) 1998-1999 Netscape Communications Corporation. All
 * Rights Reserved.
 *
 * Contributor(s):
 */
/*
 *  Copyright (c) 1995 Regents of the University of Michigan.
 *  All rights reserved.
 */
/*
 *  os-ip.c -- platform-specific TCP & UDP related code
 */

#if 0
#ifndef lint
static char copyright[] = "@(#) Copyright (c) 1995 Regents of the University of
#endif
#endif

#include "ldap-int.h"
#ifdef LDAP_CONNECT_MUST_NOT_BE_INTERRUPTED
#include <signal.h>
#endif

#ifdef NSLDAPI_HAVE_POLL
#include <poll.h>
#endif


#ifdef _WINDOWS
#define NSLDAPI_INVALID_OS_SOCKET( s )  ((s) == INVALID_SOCKET)
#else
#define NSLDAPI_INVALID_OS_SOCKET( s )  ((s) < 0 )
#endif


#define NSLDAPI_POLL_ARRAY_GROWTH  5  /* grow arrays 5 elements at a time */
```



```
/*
 * Structures and union for tracking status of network sockets
 */
#ifdef NSLDAPI_HAVE_POLL
struct nsldapi_os_statusinfo {          /* used with native OS poll() */
        struct pollfd           *ossi_pollfds;
        int                     ossi_pollfds_size;
};
______unchanged_portion_omitted_



/*
 * Non-blocking connect call function
 */
static int
nsldapi_os_connect_with_to(LBER_SOCKET sockfd, struct sockaddr *saptr,
        int salen, LDAP *ld)
{
#ifndef _WINDOWS
        int             flags;
#endif /* _WINDOWS */
        int             n, error;
        int             len;
        fd_set          rset, wset;
        struct timeval  tval;
#ifdef _WINDOWS
        int             nonblock = 1;
        int             block = 0;
        fd_set          eset;
#endif /* _WINDOWS */
        int             msec = ld->ld_connect_timeout; /* milliseconds */
        int             continue_on_intr = 0;
#ifdef _SOLARIS_SDK
        hrtime_t        start_time = 0, tmp_time, tv_time; /* nanoseconds */
#else
        long            start_time = 0, tmp_time; /* seconds */
#endif


        LDAPDebug( LDAP_DEBUG_TRACE, "nsldapi_connect_nonblock timeout: %d (msec
                msec, 0, 0);

#ifdef _WINDOWS
        ioctlsocket(sockfd, FIONBIO, &nonblock);
#else
        flags = fcntl(sockfd, F_GETFL, 0);
        fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
#endif /* _WINDOWS */

        error = 0;
        if ((n = connect(sockfd, saptr, salen)) < 0)
#ifdef _WINDOWS
                if ((n != SOCKET_ERROR) &&  (WSAGetLastError() != WSAEWOULDBLOCK
#else
                if (errno != EINPROGRESS) {
#endif /* _WINDOWS */
#ifdef LDAP_DEBUG
                        if ( ldap_debug & LDAP_DEBUG_TRACE ) {
                                perror("connect");
                        }
#endif
                        return (-1);
                }
```

```
294         /* success */
295         if (n == 0)
296                 goto done;

298         FD_ZERO(&rset);
299         FD_SET(sockfd, &rset);
300         wset = rset;

302 #ifdef _WINDOWS
303         eset = rset;
304 #endif /* _WINDOWS */

306         if (msec < 0 && msec != LDAP_X_IO_TIMEOUT_NO_TIMEOUT) {
307                 LDAPDebug( LDAP_DEBUG_TRACE, "Invalid timeout value detected.."
308                         "resetting connect timeout to default value "
309                         "(LDAP_X_IO_TIMEOUT_NO_TIMEOUT\n", 0, 0, 0);
310                 msec = LDAP_X_IO_TIMEOUT_NO_TIMEOUT;
311         } else {
312                 if (msec != 0) {
313                         tval.tv_sec = msec / MILLISEC;
314                         tval.tv_usec = (MICROSEC / MILLISEC) *
315                                         (msec % MILLISEC);
316 #ifdef _SOLARIS_SDK
317                         start_time = gethrtime();
318                         tv_time = MSEC2NSEC(msec);
320                         tv_time = (hrtime_t)msec * (NANOSEC / MILLISEC);
319 #else
320                         start_time = (long)time(NULL);
321 #endif
322                 } else {
323                         tval.tv_sec = 0;
324                         tval.tv_usec = 0;
325                 }
326         }

328         /* if timeval structure == NULL, select will block indefinitely */
329         /*                      != NULL, and value == 0, select will */
330         /*                               not block */
331         /* Windows is a bit quirky on how it behaves w.r.t nonblocking */
332         /* connects.  If the connect fails, the exception fd, eset, is */
333         /* set to show the failure.  The first argument in select is */
334         /* ignored */

336 #ifdef _WINDOWS
337         if ((n = select(sockfd +1, &rset, &wset, &eset,
338                 (msec != LDAP_X_IO_TIMEOUT_NO_TIMEOUT) ? &tval : NULL)) == 0) {
339                 errno = WSAETIMEDOUT;
340                 return (-1);
341         }
342         /* if wset is set, the connect worked */
343         if (FD_ISSET(sockfd, &wset) || FD_ISSET(sockfd, &rset)) {
344                 len = sizeof(error);
345                 if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, (char *)&error, &le
346                         < 0)
347                         return (-1);
348                 goto done;
349         }

351         /* if eset is set, the connect failed */
352         if (FD_ISSET(sockfd, &eset)) {
353                 return (-1);
354         }

356         /* failure on select call */
357         if (n == SOCKET_ERROR) {
358                 perror("select error: SOCKET_ERROR returned");
```



```
359                 return (-1);
360         }
361 #else
362         /*
363          * if LDAP_BITOPT_RESTART and select() is interrupted
364          * try again.
365          */
366         do {
367                 continue_on_intr = 0;
368                 if ((n = select(sockfd +1, &rset, &wset, NULL,
369                         (msec != LDAP_X_IO_TIMEOUT_NO_TIMEOUT) ? \
370                                 &tval : NULL)) == 0) {
371                         errno = ETIMEDOUT;
372                         return (-1);
373                 }
374                 if (n < 0) {
375                         if ((ld->ld_options & LDAP_BITOPT_RESTART) &&
376                             (errno == EINTR)) {
377                                 continue_on_intr = 1;
378                                 errno = 0;
379                                 FD_ZERO(&rset);
380                                 FD_SET(sockfd, &rset);
381                                 wset = rset;
382                                 /* honour the timeout */
383                                 if ((msec != LDAP_X_IO_TIMEOUT_NO_TIMEOUT) &&
384                                     (msec !=  0)) {
385 #ifdef _SOLARIS_SDK
386                                         tmp_time = gethrtime();
387                                         if ((tv_time -=
388                                             (tmp_time - start_time)) <= 0) {
389 #else
390                                         tmp_time = (long)time(NULL);
391                                         if ((tval.tv_sec -=
392                                             (tmp_time - start_time)) <= 0) {
393 #endif
394                                                 /* timeout */
395                                                 errno = ETIMEDOUT;
396                                                 return (-1);
397                                         }
398 #ifdef _SOLARIS_SDK
399                                         tval.tv_sec = tv_time / NANOSEC;
400                                         tval.tv_usec = (tv_time % NANOSEC) /
401                                                         (NANOSEC / MICROSEC);
402 #endif
403                                         start_time = tmp_time;
404                                 }
405                         } else {
406 #ifdef LDAP_DEBUG
407                                 perror("select error: ");
408 #endif
409                                 return (-1);
410                         }
411                 }
412         } while (continue_on_intr == 1);

414         if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
415                 len = sizeof(error);
416                 if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, (char *)&error, &le
417                         < 0)
418                         return (-1);
419 #ifdef LDAP_DEBUG
420         } else if ( ldap_debug & LDAP_DEBUG_TRACE ) {
421                 perror("select error: sockfd not set");
422 #endif
423         }
424 #endif /* _WINDOWS */
```

```
done:
#ifdef _WINDOWS
        ioctlsocket(sockfd, FIONBIO, &block);
#else
        fcntl(sockfd, F_SETFL, flags);
#endif /* _WINDOWS */

        if (error) {
                errno = error;
                return (-1);
        }

        return (0);
}
```

unchanged_portion_omitted_

```
**********************************************************
  438276 Mon Apr 28 16:23:07 2014
new/usr/src/uts/common/dtrace/dtrace.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2012, 2014 by Delphix. All rights reserved.
 */

/*
 * DTrace - Dynamic Tracing for Solaris
 *
 * This is the implementation of the Solaris Dynamic Tracing framework
 * (DTrace).  The user-visible interface to DTrace is described at length in
 * the "Solaris Dynamic Tracing Guide".  The interfaces between the libdtrace
 * library, the in-kernel DTrace framework, and the DTrace providers are
 * described in the block comments in the <sys/dtrace.h> header file.  The
 * internal architecture of DTrace is described in the block comments in the
 * <sys/dtrace_impl.h> header file.  The comments contained within the DTrace
 * implementation very much assume mastery of all of these sources; if one has
 * an unanswered question about the implementation, one should consult them
 * first.
 *
 * The functions here are ordered roughly as follows:
 *
 *   - Probe context functions
 *   - Probe hashing functions
 *   - Non-probe context utility functions
 *   - Matching functions
 *   - Provider-to-Framework API functions
 *   - Probe management functions
 *   - DIF object functions
 *   - Format functions
 *   - Predicate functions
 *   - ECB functions
 *   - Buffer functions
 *   - Enabling functions
 *   - DOF functions
 *   - Anonymous enabling functions
 *   - Consumer state functions
 *   - Helper functions
 *   - Hook functions
 *   - Driver cookbook functions
```



```
 *
 * Each group of functions begins with a block comment labelled the "DTrace
 * [Group] Functions", allowing one to find each block by searching forward
 * on capital-f functions.
 */
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cpuvar.h>
#include <sys/kmem.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/dtrace_impl.h>
#include <sys/atomic.h>
#include <sys/cmn_err.h>
#include <sys/mutex_impl.h>
#include <sys/rwlock_impl.h>
#include <sys/ctf_api.h>
#include <sys/panic.h>
#include <sys/priv_impl.h>
#include <sys/policy.h>
#include <sys/cred_impl.h>
#include <sys/procfs_isa.h>
#include <sys/taskq.h>
#include <sys/mkdev.h>
#include <sys/kdi.h>
#include <sys/zone.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "strtolctype.h"

/*
 * DTrace Tunable Variables
 *
 * The following variables may be tuned by adding a line to /etc/system that
 * includes both the name of the DTrace module ("dtrace") and the name of the
 * variable.  For example:
 *
 *   set dtrace:dtrace_destructive_disallow = 1
 *
 * In general, the only variables that one should be tuning this way are those
 * that affect system-wide DTrace behavior, and for which the default behavior
 * is undesirable.  Most of these variables are tunable on a per-consumer
 * basis using DTrace options, and need not be tuned on a system-wide basis.
 * When tuning these variables, avoid pathological values; while some attempt
 * is made to verify the integrity of these variables, they are not considered
 * part of the supported interface to DTrace, and they are therefore not
 * checked comprehensively.  Further, these variables should not be tuned
 * dynamically via "mdb -kw" or other means; they should only be tuned via
 * /etc/system.
 */
int             dtrace_destructive_disallow = 0;
dtrace_optval_t dtrace_nonroot_maxsize = (16 * 1024 * 1024);
size_t          dtrace_difo_maxsize = (256 * 1024);
dtrace_optval_t dtrace_dof_maxsize = (8 * 1024 * 1024);
size_t          dtrace_global_maxsize = (16 * 1024);
size_t          dtrace_actions_max = (16 * 1024);
size_t          dtrace_retain_max = 1024;
dtrace_optval_t dtrace_helper_actions_max = 1024;
dtrace_optval_t dtrace_helper_providers_max = 32;
dtrace_optval_t dtrace_dstate_defsize = (1 * 1024 * 1024);
size_t          dtrace_strsize_default = 256;
```

```
128 dtrace_optval_t dtrace_cleanrate_default = 9900990;             /* 101 hz */
129 dtrace_optval_t dtrace_cleanrate_min = 200000;                 /* 5000 hz */
130 dtrace_optval_t dtrace_cleanrate_max = (uint64_t)60 * NANOSEC;  /* 1/minute */
131 dtrace_optval_t dtrace_aggrate_default = NANOSEC;               /* 1 hz */
132 dtrace_optval_t dtrace_statusrate_default = NANOSEC;           /* 1 hz */
133 dtrace_optval_t dtrace_statusrate_max = (hrtime_t)10 * NANOSEC;  /* 6/minute */
134 dtrace_optval_t dtrace_switchrate_default = NANOSEC;           /* 1 hz */
135 dtrace_optval_t dtrace_nspec_default = 1;
136 dtrace_optval_t dtrace_specsize_default = 32 * 1024;
137 dtrace_optval_t dtrace_stackframes_default = 20;
138 dtrace_optval_t dtrace_ustackframes_default = 20;
139 dtrace_optval_t dtrace_jstackframes_default = 50;
140 dtrace_optval_t dtrace_jstackstrsize_default = 512;
141 int             dtrace_msgdsize_max = 128;
142 hrtime_t        dtrace_chill_max = MSEC2NSEC(500);              /* 500 ms */
142 hrtime_t        dtrace_chill_max = 500 * (NANOSEC / MILLISEC);  /* 500 ms */
143 hrtime_t        dtrace_chill_interval = NANOSEC;                /* 1000 ms */
144 int             dtrace_devdepth_max = 32;
145 int             dtrace_err_verbose;
146 hrtime_t        dtrace_deadman_interval = NANOSEC;
147 hrtime_t        dtrace_deadman_timeout = (hrtime_t)10 * NANOSEC;
148 hrtime_t        dtrace_deadman_user = (hrtime_t)30 * NANOSEC;
149 hrtime_t        dtrace_unregister_defunct_reap = (hrtime_t)60 * NANOSEC;

151 /*
152  * DTrace External Variables
153  *
154  * As dtrace(7D) is a kernel module, any DTrace variables are obviously
155  * available to DTrace consumers via the backtick (`) syntax.  One of these,
156  * dtrace_zero, is made deliberately so:  it is provided as a source of
157  * well-known, zero-filled memory.  While this variable is not documented,
158  * it is used by some translators as an implementation detail.
159  */
160 const char      dtrace_zero[256] = { 0 };       /* zero-filled memory */

162 /*
163  * DTrace Internal Variables
164  */
165 static dev_info_t       *dtrace_devi;           /* device info */
166 static vmem_t           *dtrace_arena;          /* probe ID arena */
167 static vmem_t           *dtrace_minor;          /* minor number arena */
168 static taskq_t          *dtrace_taskq;          /* task queue */
169 static dtrace_probe_t   **dtrace_probes;        /* array of all probes */
170 static int              dtrace_nprobes;         /* number of probes */
171 static dtrace_provider_t *dtrace_provider;      /* provider list */
172 static dtrace_meta_t    *dtrace_meta_pid;       /* user-land meta provider */
173 static int              dtrace_opens;           /* number of opens */
174 static int              dtrace_helpers;         /* number of helpers */
175 static int              dtrace_getf;            /* number of unpriv getf()s */
176 static void             *dtrace_softstate;      /* softstate pointer */
177 static dtrace_hash_t    *dtrace_bymod;          /* probes hashed by module */
178 static dtrace_hash_t    *dtrace_byfunc;         /* probes hashed by function */
179 static dtrace_hash_t    *dtrace_byname;         /* probes hashed by name */
180 static dtrace_toxrange_t *dtrace_toxrange;      /* toxic range array */
181 static int              dtrace_toxranges;       /* number of toxic ranges */
182 static int              dtrace_toxranges_max;   /* size of toxic range array */
183 static dtrace_anon_t    dtrace_anon;            /* anonymous enabling */
184 static kmem_cache_t     *dtrace_state_cache;    /* cache for dynamic state */
185 static uint64_t         dtrace_vtime_references; /* number of vtimestamp refs */
186 static kthread_t        *dtrace_panicked;       /* panicking thread */
187 static dtrace_ecb_t     *dtrace_ecb_create_cache; /* cached created ECB */
188 static dtrace_genid_t   dtrace_probegen;        /* current probe generation */
189 static dtrace_helpers_t *dtrace_deferred_pid;   /* deferred helper list */
190 static dtrace_enabling_t *dtrace_retained;      /* list of retained enablings */
191 static dtrace_genid_t   dtrace_retained_gen;    /* current retained enab gen */
192 static dtrace_dynvar_t  dtrace_dynhash_sink;    /* end of dynamic hash chains */
```



```
193 static int              dtrace_dynvar_failclean; /* dynvars failed to clean */

195 /*
196  * DTrace Locking
197  * DTrace is protected by three (relatively coarse-grained) locks:
198  *
199  * (1) dtrace_lock is required to manipulate essentially any DTrace state,
200  *     including enabling state, probes, ECBs, consumer state, helper state,
201  *     etc.  Importantly, dtrace_lock is _not_ required when in probe context;
202  *     probe context is lock-free -- synchronization is handled via the
203  *     dtrace_sync() cross call mechanism.
204  *
205  * (2) dtrace_provider_lock is required when manipulating provider state, or
206  *     when provider state must be held constant.
207  *
208  * (3) dtrace_meta_lock is required when manipulating meta provider state, or
209  *     when meta provider state must be held constant.
210  *
211  * The lock ordering between these three locks is dtrace_meta_lock before
212  * dtrace_provider_lock before dtrace_lock.  (In particular, there are
213  * several places where dtrace_provider_lock is held by the framework as it
214  * calls into the providers -- which then call back into the framework,
215  * grabbing dtrace_lock.)
216  *
217  * There are two other locks in the mix:  mod_lock and cpu_lock.  With respect
218  * to dtrace_provider_lock and dtrace_lock, cpu_lock continues its historical
219  * role as a coarse-grained lock; it is acquired before both of these locks.
220  * With respect to dtrace_meta_lock, its behavior is stranger:  cpu_lock must
221  * be acquired _between_ dtrace_meta_lock and any other DTrace locks.
222  * mod_lock is similar with respect to dtrace_provider_lock in that it must be
223  * acquired _between_ dtrace_provider_lock and dtrace_lock.
224  */
225 static kmutex_t         dtrace_lock;            /* probe state lock */
226 static kmutex_t         dtrace_provider_lock;   /* provider state lock */
227 static kmutex_t         dtrace_meta_lock;       /* meta-provider state lock */

229 /*
230  * DTrace Provider Variables
231  *
232  * These are the variables relating to DTrace as a provider (that is, the
233  * provider of the BEGIN, END, and ERROR probes).
234  */
235 static dtrace_pattr_t   dtrace_provider_attr = {
236 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
237 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
238 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
239 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
240 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
241 };
```

*unchanged_portion_omitted_*

```
**********************************************************
  448806 Mon Apr 28 16:23:08 2014
new/usr/src/uts/common/inet/ip/ip.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
______unchanged_portion_omitted_

1250 /*
1251  * icmp_inbound_v4 deals with ICMP messages that are handled by IP.
1252  * If the ICMP message is consumed by IP, i.e., it should not be delivered
1253  * to any IPPROTO_ICMP raw sockets, then it returns NULL.
1254  * Likewise, if the ICMP error is misformed (too short, etc), then it
1255  * returns NULL. The caller uses this to determine whether or not to send
1256  * to raw sockets.
1257  *
1258  * All error messages are passed to the matching transport stream.
1259  *
1260  * The following cases are handled by icmp_inbound:
1261  * 1) It needs to send a reply back and possibly delivering it
1262  *    to the "interested" upper clients.
1263  * 2) Return the mblk so that the caller can pass it to the RAW socket clients.
1264  * 3) It needs to change some values in IP only.
1265  * 4) It needs to change some values in IP and upper layers e.g TCP
1266  *    by delivering an error to the upper layers.
1267  *
1268  * We handle the above three cases in the context of IPsec in the
1269  * following way :
1270  *
1271  * 1) Send the reply back in the same way as the request came in.
1272  *    If it came in encrypted, it goes out encrypted. If it came in
1273  *    clear, it goes out in clear. Thus, this will prevent chosen
1274  *    plain text attack.
1275  * 2) The client may or may not expect things to come in secure.
1276  *    If it comes in secure, the policy constraints are checked
1277  *    before delivering it to the upper layers. If it comes in
1278  *    clear, ipsec_inbound_accept_clear will decide whether to
1279  *    accept this in clear or not. In both the cases, if the returned
1280  *    message (IP header + 8 bytes) that caused the icmp message has
1281  *    AH/ESP headers, it is sent up to AH/ESP for validation before
1282  *    sending up. If there are only 8 bytes of returned message, then
1283  *    upper client will not be notified.
1284  * 3) Check with global policy to see whether it matches the constaints.
1285  *    But this will be done only if icmp_accept_messages_in_clear is
1286  *    zero.
1287  * 4) If we need to change both in IP and ULP, then the decision taken
1288  *    while affecting the values in IP and while delivering up to TCP
1289  *    should be the same.
1290  *
1291  *      There are two cases.
1292  *
1293  *      a) If we reject data at the IP layer (ipsec_check_global_policy()
1294  *         failed), we will not deliver it to the ULP, even though they
1295  *         are *willing* to accept in *clear*. This is fine as our global
1296  *         disposition to icmp messages asks us reject the datagram.
1297  *
1298  *      b) If we accept data at the IP layer (ipsec_check_global_policy()
1299  *         succeeded or icmp_accept_messages_in_clear is 1), and not able
1300  *         to deliver it to ULP (policy failed), it can lead to
1301  *         consistency problems. The cases known at this time are
1302  *         ICMP_DESTINATION_UNREACHABLE  messages with following code
1303  *         values :
1304  *
1305  *         - ICMP_FRAGMENTATION_NEEDED : IP adapts to the new value
1306  *           and Upper layer rejects. Then the communication will
1307  *           come to a stop. This is solved by making similar decisions
1308  *           at both levels. Currently, when we are unable to deliver
```



```
1309  *           to the Upper Layer (due to policy failures) while IP has
1310  *           adjusted dce_pmtu, the next outbound datagram would
1311  *           generate a local ICMP_FRAGMENTATION_NEEDED message - which
1312  *           will be with the right level of protection. Thus the right
1313  *           value will be communicated even if we are not able to
1314  *           communicate when we get from the wire initially. But this
1315  *           assumes there would be at least one outbound datagram after
1316  *           IP has adjusted its dce_pmtu value. To make things
1317  *           simpler, we accept in clear after the validation of
1318  *           AH/ESP headers.
1319  *
1320  *         - Other ICMP ERRORS : We may not be able to deliver it to the
1321  *           upper layer depending on the level of protection the upper
1322  *           layer expects and the disposition in ipsec_inbound_accept_clear().
1323  *           ipsec_inbound_accept_clear() decides whether a given ICMP error
1324  *           should be accepted in clear when the Upper layer expects secure.
1325  *           Thus the communication may get aborted by some bad ICMP
1326  *           packets.
1327  */
1328 mblk_t *
1329 icmp_inbound_v4(mblk_t *mp, ip_recv_attr_t *ira)
1330 {
1331         icmph_t         *icmph;
1332         ipha_t          *ipha;          /* Outer header */
1333         int             ip_hdr_length;  /* Outer header length */
1334         boolean_t       interested;
1335         ipif_t          *ipif;
1336         uint32_t        ts;
1337         uint32_t        *tsp;
1338         timestruc_t     now;
1339         ill_t           *ill = ira->ira_ill;
1340         ip_stack_t      *ipst = ill->ill_ipst;
1341         zoneid_t        zoneid = ira->ira_zoneid;
1342         int             len_needed;
1343         mblk_t          *mp_ret = NULL;

1345         ipha = (ipha_t *)mp->b_rptr;

1347         BUMP_MIB(&ipst->ips_icmp_mib, icmpInMsgs);

1349         ip_hdr_length = ira->ira_ip_hdr_length;
1350         if ((mp->b_wptr - mp->b_rptr) < (ip_hdr_length + ICMPH_SIZE)) {
1351                 if (ira->ira_pktlen < (ip_hdr_length + ICMPH_SIZE)) {
1352                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsInTruncatedPkts);
1353                         ip_drop_input("ipIfStatsInTruncatedPkts", mp, ill);
1354                         freemsg(mp);
1355                         return (NULL);
1356                 }
1357                 /* Last chance to get real. */
1358                 ipha = ip_pullup(mp, ip_hdr_length + ICMPH_SIZE, ira);
1359                 if (ipha == NULL) {
1360                         BUMP_MIB(&ipst->ips_icmp_mib, icmpInErrors);
1361                         freemsg(mp);
1362                         return (NULL);
1363                 }
1364         }

1366         /* The IP header will always be a multiple of four bytes */
1367         icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1368         ip2dbg(("icmp_inbound_v4: type %d code %d\n", icmph->icmph_type,
1369             icmph->icmph_code));

1371         /*
1372          * We will set "interested" to "true" if we should pass a copy to
1373          * the transport or if we handle the packet locally.
1374          */
```

```
1375         interested = B_FALSE;
1376         switch (icmph->icmph_type) {
1377         case ICMP_ECHO_REPLY:
1378                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchoReps);
1379                 break;
1380         case ICMP_DEST_UNREACHABLE:
1381                 if (icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED)
1382                         BUMP_MIB(&ipst->ips_icmp_mib, icmpInFragNeeded);
1383                 interested = B_TRUE;    /* Pass up to transport */
1384                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInDestUnreachs);
1385                 break;
1386         case ICMP_SOURCE_QUENCH:
1387                 interested = B_TRUE;    /* Pass up to transport */
1388                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInSrcQuenchs);
1389                 break;
1390         case ICMP_REDIRECT:
1391                 if (!ipst->ips_ip_ignore_redirect)
1392                         interested = B_TRUE;
1393                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInRedirects);
1394                 break;
1395         case ICMP_ECHO_REQUEST:
1396                 /*
1397                  * Whether to respond to echo requests that come in as IP
1398                  * broadcasts or as IP multicast is subject to debate
1399                  * (what isn't?).  We aim to please, you pick it.
1400                  * Default is do it.
1401                  */
1402                 if (ira->ira_flags & IRAF_MULTICAST) {
1403                         /* multicast: respond based on tunable */
1404                         interested = ipst->ips_ip_g_resp_to_echo_mcast;
1405                 } else if (ira->ira_flags & IRAF_BROADCAST) {
1406                         /* broadcast: respond based on tunable */
1407                         interested = ipst->ips_ip_g_resp_to_echo_bcast;
1408                 } else {
1409                         /* unicast: always respond */
1410                         interested = B_TRUE;
1411                 }
1412                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInEchos);
1413                 if (!interested) {
1414                         /* We never pass these to RAW sockets */
1415                         freemsg(mp);
1416                         return (NULL);
1417                 }

1419                 /* Check db_ref to make sure we can modify the packet. */
1420                 if (mp->b_datap->db_ref > 1) {
1421                         mblk_t  *mp1;

1423                         mp1 = copymsg(mp);
1424                         freemsg(mp);
1425                         if (!mp1) {
1426                                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1427                                 return (NULL);
1428                         }
1429                         mp = mp1;
1430                         ipha = (ipha_t *)mp->b_rptr;
1431                         icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1432                 }
1433                 icmph->icmph_type = ICMP_ECHO_REPLY;
1434                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutEchoReps);
1435                 icmp_send_reply_v4(mp, ipha, icmph, ira);
1436                 return (NULL);

1438         case ICMP_ROUTER_ADVERTISEMENT:
1439         case ICMP_ROUTER_SOLICITATION:
1440                 break;
```



```
1441         case ICMP_TIME_EXCEEDED:
1442                 interested = B_TRUE;    /* Pass up to transport */
1443                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimeExcds);
1444                 break;
1445         case ICMP_PARAM_PROBLEM:
1446                 interested = B_TRUE;    /* Pass up to transport */
1447                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInParmProbs);
1448                 break;
1449         case ICMP_TIME_STAMP_REQUEST:
1450                 /* Response to Time Stamp Requests is local policy. */
1451                 if (ipst->ips_ip_g_resp_to_timestamp) {
1452                         if (ira->ira_flags & IRAF_MULTIBROADCAST)
1453                                 interested =
1454                                     ipst->ips_ip_g_resp_to_timestamp_bcast;
1455                         else
1456                                 interested = B_TRUE;
1457                 }
1458                 if (!interested) {
1459                         /* We never pass these to RAW sockets */
1460                         freemsg(mp);
1461                         return (NULL);
1462                 }

1464                 /* Make sure we have enough of the packet */
1465                 len_needed = ip_hdr_length + ICMPH_SIZE +
1466                     3 * sizeof (uint32_t);

1468                 if (mp->b_wptr - mp->b_rptr < len_needed) {
1469                         ipha = ip_pullup(mp, len_needed, ira);
1470                         if (ipha == NULL) {
1471                                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1472                                 ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1473                                     mp, ill);
1474                                 freemsg(mp);
1475                                 return (NULL);
1476                         }
1477                         /* Refresh following the pullup. */
1478                         icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1479                 }
1480                 BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestamps);
1481                 /* Check db_ref to make sure we can modify the packet. */
1482                 if (mp->b_datap->db_ref > 1) {
1483                         mblk_t  *mp1;

1485                         mp1 = copymsg(mp);
1486                         freemsg(mp);
1487                         if (!mp1) {
1488                                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1489                                 return (NULL);
1490                         }
1491                         mp = mp1;
1492                         ipha = (ipha_t *)mp->b_rptr;
1493                         icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1494                 }
1495                 icmph->icmph_type = ICMP_TIME_STAMP_REPLY;
1496                 tsp = (uint32_t *)&icmph[1];
1497                 tsp++;          /* Skip past 'originate time' */
1498                 /* Compute # of milliseconds since midnight */
1499                 gethrestime(&now);
1500                 ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
1501                     NSEC2MSEC(now.tv_nsec);
1501                     now.tv_nsec / (NANOSEC / MILLISEC);
1502                 *tsp++ = htonl(ts);     /* Lay in 'receive time' */
1503                 *tsp++ = htonl(ts);     /* Lay in 'send time' */
1504                 BUMP_MIB(&ipst->ips_icmp_mib, icmpOutTimestampReps);
1505                 icmp_send_reply_v4(mp, ipha, icmph, ira);
```

```
1506                return (NULL);

1508        case ICMP_TIME_STAMP_REPLY:
1509                BUMP_MIB(&ipst->ips_icmp_mib, icmpInTimestampReps);
1510                break;
1511        case ICMP_INFO_REQUEST:
1512                /* Per RFC 1122 3.2.2.7, ignore this. */
1513        case ICMP_INFO_REPLY:
1514                break;
1515        case ICMP_ADDRESS_MASK_REQUEST:
1516                if (ira->ira_flags & IRAF_MULTIBROADCAST) {
1517                        interested =
1518                            ipst->ips_ip_respond_to_address_mask_broadcast;
1519                } else {
1520                        interested = B_TRUE;
1521                }
1522                if (!interested) {
1523                        /* We never pass these to RAW sockets */
1524                        freemsg(mp);
1525                        return (NULL);
1526                }
1527                len_needed = ip_hdr_length + ICMPH_SIZE + IP_ADDR_LEN;
1528                if (mp->b_wptr - mp->b_rptr < len_needed) {
1529                        ipha = ip_pullup(mp, len_needed, ira);
1530                        if (ipha == NULL) {
1531                                BUMP_MIB(ill->ill_ip_mib,
1532                                    ipIfStatsInTruncatedPkts);
1533                                ip_drop_input("ipIfStatsInTruncatedPkts", mp,
1534                                    ill);
1535                                freemsg(mp);
1536                                return (NULL);
1537                        }
1538                        /* Refresh following the pullup. */
1539                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1540                }
1541                BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMasks);
1542                /* Check db_ref to make sure we can modify the packet. */
1543                if (mp->b_datap->db_ref > 1) {
1544                        mblk_t  *mp1;

1546                        mp1 = copymsg(mp);
1547                        freemsg(mp);
1548                        if (!mp1) {
1549                                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutDrops);
1550                                return (NULL);
1551                        }
1552                        mp = mp1;
1553                        ipha = (ipha_t *)mp->b_rptr;
1554                        icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1555                }
1556                /*
1557                 * Need the ipif with the mask be the same as the source
1558                 * address of the mask reply. For unicast we have a specific
1559                 * ipif. For multicast/broadcast we only handle onlink
1560                 * senders, and use the source address to pick an ipif.
1561                 */
1562                ipif = ipif_lookup_addr(ipha->ipha_dst, ill, zoneid, ipst);
1563                if (ipif == NULL) {
1564                        /* Broadcast or multicast */
1565                        ipif = ipif_lookup_remote(ill, ipha->ipha_src, zoneid);
1566                        if (ipif == NULL) {
1567                                freemsg(mp);
1568                                return (NULL);
1569                        }
1570                }
1571                icmph->icmph_type = ICMP_ADDRESS_MASK_REPLY;
```



```
1572                bcopy(&ipif->ipif_net_mask, &icmph[1], IP_ADDR_LEN);
1573                ipif_refrele(ipif);
1574                BUMP_MIB(&ipst->ips_icmp_mib, icmpOutAddrMaskReps);
1575                icmp_send_reply_v4(mp, ipha, icmph, ira);
1576                return (NULL);

1578        case ICMP_ADDRESS_MASK_REPLY:
1579                BUMP_MIB(&ipst->ips_icmp_mib, icmpInAddrMaskReps);
1580                break;
1581        default:
1582                interested = B_TRUE;    /* Pass up to transport */
1583                BUMP_MIB(&ipst->ips_icmp_mib, icmpInUnknowns);
1584                break;
1585        }
1586        /*
1587         * See if there is an ICMP client to avoid an extra copymsg/freemsg
1588         * if there isn't one.
1589         */
1590        if (ipst->ips_ipcl_proto_fanout_v4[IPPROTO_ICMP].connf_head != NULL) {
1591                /* If there is an ICMP client and we want one too, copy it. */

1593                if (!interested) {
1594                        /* Caller will deliver to RAW sockets */
1595                        return (mp);
1596                }
1597                mp_ret = copymsg(mp);
1598                if (mp_ret == NULL) {
1599                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1600                        ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1601                }
1602        } else if (!interested) {
1603                /* Neither we nor raw sockets are interested. Drop packet now */
1604                freemsg(mp);
1605                return (NULL);
1606        }

1608        /*
1609         * ICMP error or redirect packet. Make sure we have enough of
1610         * the header and that db_ref == 1 since we might end up modifying
1611         * the packet.
1612         */
1613        if (mp->b_cont != NULL) {
1614                if (ip_pullup(mp, -1, ira) == NULL) {
1615                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1616                        ip_drop_input("ipIfStatsInDiscards - ip_pullup",
1617                            mp, ill);
1618                        freemsg(mp);
1619                        return (mp_ret);
1620                }
1621        }

1623        if (mp->b_datap->db_ref > 1) {
1624                mblk_t  *mp1;

1626                mp1 = copymsg(mp);
1627                if (mp1 == NULL) {
1628                        BUMP_MIB(ill->ill_ip_mib, ipIfStatsInDiscards);
1629                        ip_drop_input("ipIfStatsInDiscards - copymsg", mp, ill);
1630                        freemsg(mp);
1631                        return (mp_ret);
1632                }
1633                freemsg(mp);
1634                mp = mp1;
1635        }

1637        /*
```

```
1638          * In case mp has changed, verify the message before any further
1639          * processes.
1640          */
1641         ipha = (ipha_t *)mp->b_rptr;
1642         icmph = (icmph_t *)&mp->b_rptr[ip_hdr_length];
1643         if (!icmp_inbound_verify_v4(mp, icmph, ira)) {
1644                 freemsg(mp);
1645                 return (mp_ret);
1646         }

1648         switch (icmph->icmph_type) {
1649         case ICMP_REDIRECT:
1650                 icmp_redirect_v4(mp, ipha, icmph, ira);
1651                 break;
1652         case ICMP_DEST_UNREACHABLE:
1653                 if (icmph->icmph_code == ICMP_FRAGMENTATION_NEEDED) {
1654                         /* Update DCE and adjust MTU is icmp header if needed */
1655                         icmp_inbound_too_big_v4(icmph, ira);
1656                 }
1657                 /* FALLTHRU */
1658         default:
1659                 icmp_inbound_error_fanout_v4(mp, icmph, ira);
1660                 break;
1661         }
1662         return (mp_ret);
1663 }
```

unchanged_portion_omitted

```
8958 /*
8959  * Update any source route, record route or timestamp options
8960  * When it fails it has consumed the message and BUMPed the MIB.
8961  */
8962 boolean_t
8963 ip_forward_options(mblk_t *mp, ipha_t *ipha, ill_t *dst_ill,
8964     ip_recv_attr_t *ira)
8965 {
8966         ipoptp_t        opts;
8967         uchar_t         *opt;
8968         uint8_t         optval;
8969         uint8_t         optlen;
8970         ipaddr_t        dst;
8971         ipaddr_t        ifaddr;
8972         uint32_t        ts;
8973         timestruc_t     now;
8974         ip_stack_t      *ipst = ira->ira_ill->ill_ipst;

8976         ip2dbg(("ip_forward_options\n"));
8977         dst = ipha->ipha_dst;
8978         for (optval = ipoptp_first(&opts, ipha);
8979             optval != IPOPT_EOL;
8980             optval = ipoptp_next(&opts)) {
8981                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
8982                 opt = opts.ipoptp_cur;
8983                 optlen = opts.ipoptp_len;
8984                 ip2dbg(("ip_forward_options: opt %d, len %d\n",
8985                     optval, opts.ipoptp_len));
8986                 switch (optval) {
8987                         uint32_t off;
8988                 case IPOPT_SSRR:
8989                 case IPOPT_LSRR:
8990                         /* Check if adminstratively disabled */
8991                         if (!ipst->ips_ip_forward_src_routed) {
8992                                 BUMP_MIB(dst_ill->ill_ip_mib,
8993                                     ipIfStatsForwProhibits);
8994                                 ip_drop_input("ICMP_SOURCE_ROUTE_FAILED",
8995                                     mp, dst_ill);
```



```
8996                                 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED,
8997                                     ira);
8998                                 return (B_FALSE);
8999                         }
9000                         if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9001                                 /*
9002                                  * Must be partial since ip_input_options
9003                                  * checked for strict.
9004                                  */
9005                                 break;
9006                         }
9007                         off = opt[IPOPT_OFFSET];
9008                         off--;
9009                 redo_srr:
9010                         if (optlen < IP_ADDR_LEN ||
9011                             off > optlen - IP_ADDR_LEN) {
9012                                 /* End of source route */
9013                                 ip1dbg((
9014                                     "ip_forward_options: end of SR\n"));
9015                                 break;
9016                         }
9017                         /* Pick a reasonable address on the outbound if */
9018                         ASSERT(dst_ill != NULL);
9019                         if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9020                             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9021                             NULL) != 0) {
9022                                 /* No source! Shouldn't happen */
9023                                 ifaddr = INADDR_ANY;
9024                         }
9025                         bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9026                         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9027                         ip1dbg(("ip_forward_options: next hop 0x%x\n",
9028                             ntohl(dst)));

9030                         /*
9031                          * Check if our address is present more than
9032                          * once as consecutive hops in source route.
9033                          */
9034                         if (ip_type_v4(dst, ipst) == IRE_LOCAL) {
9035                                 off += IP_ADDR_LEN;
9036                                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9037                                 goto redo_srr;
9038                         }
9039                         ipha->ipha_dst = dst;
9040                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9041                         break;
9042                 case IPOPT_RR:
9043                         off = opt[IPOPT_OFFSET];
9044                         off--;
9045                         if (optlen < IP_ADDR_LEN ||
9046                             off > optlen - IP_ADDR_LEN) {
9047                                 /* No more room - ignore */
9048                                 ip1dbg((
9049                                     "ip_forward_options: end of RR\n"));
9050                                 break;
9051                         }
9052                         /* Pick a reasonable address on the outbound if */
9053                         ASSERT(dst_ill != NULL);
9054                         if (ip_select_source_v4(dst_ill, INADDR_ANY, dst,
9055                             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9056                             NULL) != 0) {
9057                                 /* No source! Shouldn't happen */
9058                                 ifaddr = INADDR_ANY;
9059                         }
9060                         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9061                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
```

```
9062                    break;
9063            case IPOPT_TS:
9064                    /* Insert timestamp if there is room */
9065                    switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9066                    case IPOPT_TS_TSONLY:
9067                            off = IPOPT_TS_TIMELEN;
9068                            break;
9069                    case IPOPT_TS_PRESPEC:
9070                    case IPOPT_TS_PRESPEC_RFC791:
9071                            /* Verify that the address matched */
9072                            off = opt[IPOPT_OFFSET] - 1;
9073                            bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9074                            if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9075                                    /* Not for us */
9076                                    break;
9077                            }
9078                            /* FALLTHRU */
9079                    case IPOPT_TS_TSANDADDR:
9080                            off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9081                            break;
9082                    default:
9083                            /*
9084                             * ip_*put_options should have already
9085                             * dropped this packet.
9086                             */
9087                            cmn_err(CE_PANIC, "ip_forward_options: "
9088                                "unknown IT - bug in ip_input_options?\n");
9089                            return (B_TRUE);        /* Keep "lint" happy */
9090                    }
9091                    if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9092                            /* Increase overflow counter */
9093                            off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9094                            opt[IPOPT_POS_OV_FLG] =
9095                                (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9096                                (off << 4));
9097                            break;
9098                    }
9099                    off = opt[IPOPT_OFFSET] - 1;
9100                    switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9101                    case IPOPT_TS_PRESPEC:
9102                    case IPOPT_TS_PRESPEC_RFC791:
9103                    case IPOPT_TS_TSANDADDR:
9104                            /* Pick a reasonable addr on the outbound if */
9105                            ASSERT(dst_ill != NULL);
9106                            if (ip_select_source_v4(dst_ill, INADDR_ANY,
9107                                dst, INADDR_ANY, ALL_ZONES, ipst, &ifaddr,
9108                                NULL, NULL) != 0) {
9109                                    /* No source! Shouldn't happen */
9110                                    ifaddr = INADDR_ANY;
9111                            }
9112                            bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9113                            opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9114                            /* FALLTHRU */
9115                    case IPOPT_TS_TSONLY:
9116                            off = opt[IPOPT_OFFSET] - 1;
9117                            /* Compute # of milliseconds since midnight */
9118                            gethrestime(&now);
9119                            ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9120                                NSEC2MSEC(now.tv_nsec);
9120                                now.tv_nsec / (NANOSEC / MILLISEC);
9121                            bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9122                            opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9123                            break;
9124                    }
9125                    break;
9126            }
```



```
9127            }
9128            return (B_TRUE);
9129    }
_______unchanged_portion_omitted_

9209    /*
9210     * Update any source route, record route or timestamp options.
9211     * Check that we are at end of strict source route.
9212     * The options have already been checked for sanity in ip_input_options().
9213     */
9214    boolean_t
9215    ip_input_local_options(mblk_t *mp, ipha_t *ipha, ip_recv_attr_t *ira)
9216    {
9217            ipoptp_t        opts;
9218            uchar_t         *opt;
9219            uint8_t         optval;
9220            uint8_t         optlen;
9221            ipaddr_t        dst;
9222            ipaddr_t        ifaddr;
9223            uint32_t        ts;
9224            timestruc_t     now;
9225            ill_t           *ill = ira->ira_ill;
9226            ip_stack_t      *ipst = ill->ill_ipst;

9228            ip2dbg(("ip_input_local_options\n"));

9230            for (optval = ipoptp_first(&opts, ipha);
9231                optval != IPOPT_EOL;
9232                optval = ipoptp_next(&opts)) {
9233                    ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
9234                    opt = opts.ipoptp_cur;
9235                    optlen = opts.ipoptp_len;
9236                    ip2dbg(("ip_input_local_options: opt %d, len %d\n",
9237                        optval, optlen));
9238                    switch (optval) {
9239                            uint32_t off;
9240                    case IPOPT_SSRR:
9241                    case IPOPT_LSRR:
9242                            off = opt[IPOPT_OFFSET];
9243                            off--;
9244                            if (optlen < IP_ADDR_LEN ||
9245                                off > optlen - IP_ADDR_LEN) {
9246                                    /* End of source route */
9247                                    ip1dbg(("ip_input_local_options: end of SR\n"));
9248                                    break;
9249                            }
9250                            /*
9251                             * This will only happen if two consecutive entries
9252                             * in the source route contains our address or if
9253                             * it is a packet with a loose source route which
9254                             * reaches us before consuming the whole source route
9255                             */
9256                            ip1dbg(("ip_input_local_options: not end of SR\n"));
9257                            if (optval == IPOPT_SSRR) {
9258                                    goto bad_src_route;
9259                            }
9260                            /*
9261                             * Hack: instead of dropping the packet truncate the
9262                             * source route to what has been used by filling the
9263                             * rest with IPOPT_NOP.
9264                             */
9265                            opt[IPOPT_OLEN] = (uint8_t)off;
9266                            while (off < optlen) {
9267                                    opt[off++] = IPOPT_NOP;
9268                            }
9269                            break;
```

```
9270                 case IPOPT_RR:
9271                         off = opt[IPOPT_OFFSET];
9272                         off--;
9273                         if (optlen < IP_ADDR_LEN ||
9274                             off > optlen - IP_ADDR_LEN) {
9275                                 /* No more room - ignore */
9276                                 ip1dbg((
9277                                     "ip_input_local_options: end of RR\n"));
9278                                 break;
9279                         }
9280                         /* Pick a reasonable address on the outbound if */
9281                         if (ip_select_source_v4(ill, INADDR_ANY, ipha->ipha_dst,
9282                             INADDR_ANY, ALL_ZONES, ipst, &ifaddr, NULL,
9283                             NULL) != 0) {
9284                                 /* No source! Shouldn't happen */
9285                                 ifaddr = INADDR_ANY;
9286                         }
9287                         bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9288                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9289                         break;
9290                 case IPOPT_TS:
9291                         /* Insert timestamp if there is romm */
9292                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9293                         case IPOPT_TS_TSONLY:
9294                                 off = IPOPT_TS_TIMELEN;
9295                                 break;
9296                         case IPOPT_TS_PRESPEC:
9297                         case IPOPT_TS_PRESPEC_RFC791:
9298                                 /* Verify that the address matched */
9299                                 off = opt[IPOPT_OFFSET] - 1;
9300                                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
9301                                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
9302                                         /* Not for us */
9303                                         break;
9304                                 }
9305                                 /* FALLTHRU */
9306                         case IPOPT_TS_TSANDADDR:
9307                                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
9308                                 break;
9309                         default:
9310                                 /*
9311                                  * ip_*put_options should have already
9312                                  * dropped this packet.
9313                                  */
9314                                 cmn_err(CE_PANIC, "ip_input_local_options: "
9315                                     "unknown IT - bug in ip_input_options?\n");
9316                                 return (B_TRUE);        /* Keep "lint" happy */
9317                         }
9318                         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
9319                                 /* Increase overflow counter */
9320                                 off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
9321                                 opt[IPOPT_POS_OV_FLG] =
9322                                     (uint8_t)((opt[IPOPT_POS_OV_FLG] & 0x0F) |
9323                                     (off << 4));
9324                                 break;
9325                         }
9326                         off = opt[IPOPT_OFFSET] - 1;
9327                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
9328                         case IPOPT_TS_PRESPEC:
9329                         case IPOPT_TS_PRESPEC_RFC791:
9330                         case IPOPT_TS_TSANDADDR:
9331                                 /* Pick a reasonable addr on the outbound if */
9332                                 if (ip_select_source_v4(ill, INADDR_ANY,
9333                                     ipha->ipha_dst, INADDR_ANY, ALL_ZONES, ipst,
9334                                     &ifaddr, NULL, NULL) != 0) {
9335                                         /* No source! Shouldn't happen */
```



```
9336                                         ifaddr = INADDR_ANY;
9337                                 }
9338                                 bcopy(&ifaddr, (char *)opt + off, IP_ADDR_LEN);
9339                                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
9340                                 /* FALLTHRU */
9341                         case IPOPT_TS_TSONLY:
9342                                 off = opt[IPOPT_OFFSET] - 1;
9343                                 /* Compute # of milliseconds since midnight */
9344                                 gethrestime(&now);
9345                                 ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
9346                                     NSEC2MSEC(now.tv_nsec);
9346                                     now.tv_nsec / (NANOSEC / MILLISEC);
9347                                 bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
9348                                 opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
9349                                 break;
9350                         }
9351                         break;
9352                 }
9353         }
9354         return (B_TRUE);

9356 bad_src_route:
9357         /* make sure we clear any indication of a hardware checksum */
9358         DB_CKSUMFLAGS(mp) = 0;
9359         ip_drop_input("ICMP_SOURCE_ROUTE_FAILED", mp, ill);
9360         icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, ira);
9361         return (B_FALSE);

9363 }
______unchanged_portion_omitted_

11910 /*
11911  * Update any source route, record route, or timestamp options when
11912  * sending a packet back to ourselves.
11913  * Check that we are at end of strict source route.
11914  * The options have been sanity checked by ip_output_options().
11915  */
11916 void
11917 ip_output_local_options(ipha_t *ipha, ip_stack_t *ipst)
11918 {
11919         ipoptp_t        opts;
11920         uchar_t         *opt;
11921         uint8_t         optval;
11922         uint8_t         optlen;
11923         ipaddr_t        dst;
11924         uint32_t        ts;
11925         timestruc_t     now;

11927         for (optval = ipoptp_first(&opts, ipha);
11928             optval != IPOPT_EOL;
11929             optval = ipoptp_next(&opts)) {
11930                 opt = opts.ipoptp_cur;
11931                 optlen = opts.ipoptp_len;
11932                 ASSERT((opts.ipoptp_flags & IPOPTP_ERROR) == 0);
11933                 switch (optval) {
11934                         uint32_t off;
11935                 case IPOPT_SSRR:
11936                 case IPOPT_LSRR:
11937                         off = opt[IPOPT_OFFSET];
11938                         off--;
11939                         if (optlen < IP_ADDR_LEN ||
11940                             off > optlen - IP_ADDR_LEN) {
11941                                 /* End of source route */
11942                                 break;
11943                         }
11944                         /*
```

```
11945                          * This will only happen if two consecutive entries
11946                          * in the source route contains our address or if
11947                          * it is a packet with a loose source route which
11948                          * reaches us before consuming the whole source route
11949                          */

11951                         if (optval == IPOPT_SSRR) {
11952                                 return;
11953                         }
11954                         /*
11955                          * Hack: instead of dropping the packet truncate the
11956                          * source route to what has been used by filling the
11957                          * rest with IPOPT_NOP.
11958                          */
11959                         opt[IPOPT_OLEN] = (uint8_t)off;
11960                         while (off < optlen) {
11961                                 opt[off++] = IPOPT_NOP;
11962                         }
11963                         break;
11964                 case IPOPT_RR:
11965                         off = opt[IPOPT_OFFSET];
11966                         off--;
11967                         if (optlen < IP_ADDR_LEN ||
11968                             off > optlen - IP_ADDR_LEN) {
11969                                 /* No more room - ignore */
11970                                 ip1dbg((
11971                                     "ip_output_local_options: end of RR\n"));
11972                                 break;
11973                         }
11974                         dst = htonl(INADDR_LOOPBACK);
11975                         bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
11976                         opt[IPOPT_OFFSET] += IP_ADDR_LEN;
11977                         break;
11978                 case IPOPT_TS:
11979                         /* Insert timestamp if there is romm */
11980                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
11981                         case IPOPT_TS_TSONLY:
11982                                 off = IPOPT_TS_TIMELEN;
11983                                 break;
11984                         case IPOPT_TS_PRESPEC:
11985                         case IPOPT_TS_PRESPEC_RFC791:
11986                                 /* Verify that the address matched */
11987                                 off = opt[IPOPT_OFFSET] - 1;
11988                                 bcopy((char *)opt + off, &dst, IP_ADDR_LEN);
11989                                 if (ip_type_v4(dst, ipst) != IRE_LOCAL) {
11990                                         /* Not for us */
11991                                         break;
11992                                 }
11993                                 /* FALLTHRU */
11994                         case IPOPT_TS_TSANDADDR:
11995                                 off = IP_ADDR_LEN + IPOPT_TS_TIMELEN;
11996                                 break;
11997                         default:
11998                                 /*
11999                                  * ip_*put_options should have already
12000                                  * dropped this packet.
12001                                  */
12002                                 cmn_err(CE_PANIC, "ip_output_local_options: "
12003                                     "unknown IT - bug in ip_output_options?\n");
12004                                 return; /* Keep "lint" happy */
12005                         }
12006                         if (opt[IPOPT_OFFSET] - 1 + off > optlen) {
12007                                 /* Increase overflow counter */
12008                                 off = (opt[IPOPT_POS_OV_FLG] >> 4) + 1;
12009                                 opt[IPOPT_POS_OV_FLG] = (uint8_t)
12010                                     (opt[IPOPT_POS_OV_FLG] & 0x0F) |
```



```
12011                                     (off << 4);
12012                                 break;
12013                         }
12014                         off = opt[IPOPT_OFFSET] - 1;
12015                         switch (opt[IPOPT_POS_OV_FLG] & 0x0F) {
12016                         case IPOPT_TS_PRESPEC:
12017                         case IPOPT_TS_PRESPEC_RFC791:
12018                         case IPOPT_TS_TSANDADDR:
12019                                 dst = htonl(INADDR_LOOPBACK);
12020                                 bcopy(&dst, (char *)opt + off, IP_ADDR_LEN);
12021                                 opt[IPOPT_OFFSET] += IP_ADDR_LEN;
12022                                 /* FALLTHRU */
12023                         case IPOPT_TS_TSONLY:
12024                                 off = opt[IPOPT_OFFSET] - 1;
12025                                 /* Compute # of milliseconds since midnight */
12026                                 gethrestime(&now);
12027                                 ts = (now.tv_sec % (24 * 60 * 60)) * 1000 +
12028                                     NSEC2MSEC(now.tv_nsec);
12028                                     now.tv_nsec / (NANOSEC / MILLISEC);
12029                                 bcopy(&ts, (char *)opt + off, IPOPT_TS_TIMELEN);
12030                                 opt[IPOPT_OFFSET] += IPOPT_TS_TIMELEN;
12031                                 break;
12032                         }
12033                         break;
12034                 }
12035         }
12036 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   24657 Mon Apr 28 16:23:09 2014
new/usr/src/uts/common/io/devpoll.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
______unchanged_portion_omitted_

 693 /*ARGSUSED*/
 694 static int
 695 dpioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp, int *rvalp)
 696 {
 697         minor_t         minor;
 698         dp_entry_t      *dpep;
 699         pollcache_t     *pcp;
 700         hrtime_t        now;
 701         int             error = 0;
 702         STRUCT_DECL(dvpoll, dvpoll);

 704         if (cmd == DP_POLL) {
 705                 /* do this now, before we sleep on DP_WRITER_PRESENT */
 706                 now = gethrtime();
 707         }

 709         minor = getminor(dev);
 710         mutex_enter(&devpoll_lock);
 711         ASSERT(minor < dptblsize);
 712         dpep = devpolltbl[minor];
 713         mutex_exit(&devpoll_lock);
 714         ASSERT(dpep != NULL);
 715         pcp = dpep->dpe_pcache;
 716         if (curproc->p_pid != pcp->pc_pid)
 717                 return (EACCES);

 719         mutex_enter(&dpep->dpe_lock);
 720         while ((dpep->dpe_flag & DP_WRITER_PRESENT) ||
 721             (dpep->dpe_writerwait != 0)) {
 722                 if (!cv_wait_sig_swap(&dpep->dpe_cv, &dpep->dpe_lock)) {
 723                         mutex_exit(&dpep->dpe_lock);
 724                         return (EINTR);
 725                 }
 726         }
 727         dpep->dpe_refcnt++;
 728         mutex_exit(&dpep->dpe_lock);

 730         switch (cmd) {
 731         case    DP_POLL:
 732         {
 733                 pollstate_t     *ps;
 734                 nfds_t          nfds;
 735                 int             fdcnt = 0;
 736                 hrtime_t        deadline = 0;

 738                 STRUCT_INIT(dvpoll, mode);
 739                 error = copyin((caddr_t)arg, STRUCT_BUF(dvpoll),
 740                     STRUCT_SIZE(dvpoll));
 741                 if (error) {
 742                         DP_REFRELE(dpep);
 743                         return (EFAULT);
 744                 }

 746                 deadline = STRUCT_FGET(dvpoll, dp_timeout);
 747                 if (deadline > 0) {
 748                         /*
 749                          * Convert the deadline from relative milliseconds
 750                          * to absolute nanoseconds.  They must wait for at
 751                          * least a tick.
```



```
 752                          */
 753                         deadline = MSEC2NSEC(deadline);
 753                         deadline = deadline * NANOSEC / MILLISEC;
 754                         deadline = MAX(deadline, nsec_per_tick);
 755                         deadline += now;
 756                 }

 758                 if ((nfds = STRUCT_FGET(dvpoll, dp_nfds)) == 0) {
 759                         /*
 760                          * We are just using DP_POLL to sleep, so
 761                          * we don't any of the devpoll apparatus.
 762                          * Do not check for signals if we have a zero timeout.
 763                          */
 764                         DP_REFRELE(dpep);
 765                         if (deadline == 0)
 766                                 return (0);
 767                         mutex_enter(&curthread->t_delay_lock);
 768                         while ((error =
 769                             cv_timedwait_sig_hrtime(&curthread->t_delay_cv,
 770                             &curthread->t_delay_lock, deadline)) > 0)
 771                                 continue;
 772                         mutex_exit(&curthread->t_delay_lock);
 773                         return (error == 0 ? EINTR : 0);
 774                 }

 776                 /*
 777                  * XXX It would be nice not to have to alloc each time, but it
 778                  * requires another per thread structure hook. This can be
 779                  * implemented later if data suggests that it's necessary.
 780                  */
 781                 if ((ps = curthread->t_pollstate) == NULL) {
 782                         curthread->t_pollstate = pollstate_create();
 783                         ps = curthread->t_pollstate;
 784                 }
 785                 if (ps->ps_dpbufsize < nfds) {
 786                         struct proc *p = ttoproc(curthread);
 787                         /*
 788                          * The maximum size should be no large than
 789                          * current maximum open file count.
 790                          */
 791                         mutex_enter(&p->p_lock);
 792                         if (nfds > p->p_fno_ctl) {
 793                                 mutex_exit(&p->p_lock);
 794                                 DP_REFRELE(dpep);
 795                                 return (EINVAL);
 796                         }
 797                         mutex_exit(&p->p_lock);
 798                         kmem_free(ps->ps_dpbuf, sizeof (pollfd_t) *
 799                             ps->ps_dpbufsize);
 800                         ps->ps_dpbuf = kmem_zalloc(sizeof (pollfd_t) *
 801                             nfds, KM_SLEEP);
 802                         ps->ps_dpbufsize = nfds;
 803                 }

 805                 mutex_enter(&pcp->pc_lock);
 806                 for (;;) {
 807                         pcp->pc_flag = 0;
 808                         error = dp_pcache_poll(ps->ps_dpbuf, pcp, nfds, &fdcnt);
 809                         if (fdcnt > 0 || error != 0)
 810                                 break;

 812                         /*
 813                          * A pollwake has happened since we polled cache.
 814                          */
 815                         if (pcp->pc_flag & T_POLLWAKE)
 816                                 continue;
```

```
818                         /*
819                          * Sleep until we are notified, signaled, or timed out.
820                          */
821                         if (deadline == 0) {
822                                 /* immediate timeout; do not check signals */
823                                 break;
824                         }
825                         error = cv_timedwait_sig_hrtime(&pcp->pc_cv,
826                             &pcp->pc_lock, deadline);
827                         /*
828                          * If we were awakened by a signal or timeout
829                          * then break the loop, else poll again.
830                          */
831                         if (error <= 0) {
832                                 error = (error == 0) ? EINTR : 0;
833                                 break;
834                         } else {
835                                 error = 0;
836                         }
837                 }
838                 mutex_exit(&pcp->pc_lock);

840                 if (error == 0 && fdcnt > 0) {
841                         if (copyout(ps->ps_dpbuf, STRUCT_FGETP(dvpoll,
842                             dp_fds), sizeof (pollfd_t) * fdcnt)) {
843                                 DP_REFRELE(dpep);
844                                 return (EFAULT);
845                         }
846                         *rvalp = fdcnt;
847                 }
848                 break;
849         }

851         case    DP_ISPOLLED:
852         {
853                 pollfd_t        pollfd;
854                 polldat_t       *pdp;

856                 STRUCT_INIT(dvpoll, mode);
857                 error = copyin((caddr_t)arg, &pollfd, sizeof (pollfd_t));
858                 if (error) {
859                         DP_REFRELE(dpep);
860                         return (EFAULT);
861                 }
862                 mutex_enter(&pcp->pc_lock);
863                 if (pcp->pc_hash == NULL) {
864                         /*
865                          * No Need to search because no poll fd
866                          * has been cached.
867                          */
868                         mutex_exit(&pcp->pc_lock);
869                         DP_REFRELE(dpep);
870                         return (0);
871                 }
872                 if (pollfd.fd < 0) {
873                         mutex_exit(&pcp->pc_lock);
874                         break;
875                 }
876                 pdp = pcache_lookup_fd(pcp, pollfd.fd);
877                 if ((pdp != NULL) && (pdp->pd_fd == pollfd.fd) &&
878                     (pdp->pd_fp != NULL)) {
879                         pollfd.revents = pdp->pd_events;
880                         if (copyout(&pollfd, (caddr_t)arg, sizeof (pollfd_t))) {
881                                 mutex_exit(&pcp->pc_lock);
882                                 DP_REFRELE(dpep);
```



```
883                                 return (EFAULT);
884                         }
885                         *rvalp = 1;
886                 }
887                 mutex_exit(&pcp->pc_lock);
888                 break;
889         }

891         default:
892                 DP_REFRELE(dpep);
893                 return (EINVAL);
894         }
895         DP_REFRELE(dpep);
896         return (error);
897 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   32137 Mon Apr 28 16:23:09 2014
new/usr/src/uts/common/io/power.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright 2011 Joyent, Inc.  All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 *      Power Button Driver
 *
 *      This driver handles interrupt generated by the power button on
 *      platforms with "power" device node which has "button" property.
 *      Currently, these platforms are:
 *
 *              ACPI-enabled x86/x64 platforms
 *              Ultra-5_10, Ultra-80, Sun-Blade-100, Sun-Blade-150,
 *              Sun-Blade-1500, Sun-Blade-2500,
 *              Sun-Fire-V210, Sun-Fire-V240, Netra-240
 *
 *      Only one instance is allowed to attach.  In order to know when
 *      an application that has opened the device is going away, a new
 *      minor clone is created for each open(9E) request.  There are
 *      allocations for creating minor clones between 1 and 255.  The ioctl
 *      interface is defined by pbio(7I) and approved as part of
 *      PSARC/1999/393 case.
 */

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/cmn_err.h>
#include <sys/errno.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <sys/pbio.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/pwrctl.h>
```



```
#if defined(__sparc)
#include <sys/machsystm.h>
#endif

#ifdef  ACPI_POWER_BUTTON

#include <sys/acpi/acpi.h>
#include <sys/acpica.h>

#else

#include <sys/epic.h>
/*
 * Some #defs that must be here as they differ for power.c
 * and epic.c
 */
#define EPIC_REGS_OFFSET        0x00
#define EPIC_REGS_LEN           0x82


/*
 * This flag, which is set for platforms,  that have EPIC processor
 * to process power button interrupt, helps in executing platform
 * specific code.
 */
static char     hasEPIC = B_FALSE;
#endif  /* ACPI_POWER_BUTTON */

/*
 * Maximum number of clone minors that is allowed.  This value
 * is defined relatively low to save memory.
 */
#define POWER_MAX_CLONE 256

/*
 * Minor number is instance << 8 + clone minor from range 1-255; clone 0
 * is reserved for "original" minor.
 */
#define POWER_MINOR_TO_CLONE(minor) ((minor) & (POWER_MAX_CLONE - 1))

/*
 * Power Button Abort Delay
 */
#define ABORT_INCREMENT_DELAY   10

/*
 * FWARC 2005/687: power device compatible property
 */
#define POWER_DEVICE_TYPE "power-device-type"

/*
 * Driver global variables
 */
static void *power_state;
static int power_inst = -1;

static hrtime_t power_button_debounce = MSEC2NSEC(10);
static hrtime_t power_button_debounce = NANOSEC/MILLISEC*10;
static hrtime_t power_button_abort_interval = 1.5 * NANOSEC;
static int      power_button_abort_presses = 3;
static int      power_button_abort_enable = 1;
static int      power_button_enable = 1;

static int      power_button_pressed = 0;
static int      power_button_cancel = 0;
```

```
127 static int      power_button_timeouts = 0;
128 static int      timeout_cancel = 0;
129 static int      additional_presses = 0;

131 /*
132  * Function prototypes
133  */
134 static int power_attach(dev_info_t *, ddi_attach_cmd_t);
135 static int power_detach(dev_info_t *, ddi_detach_cmd_t);
136 static int power_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
137 static int power_open(dev_t *, int, int, cred_t *);
138 static int power_close(dev_t, int, int, cred_t *);
139 static int power_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);
140 static int power_chpoll(dev_t, short, int, short *, struct pollhead **);
141 #ifndef ACPI_POWER_BUTTON
142 static uint_t power_high_intr(caddr_t);
143 #endif
144 static uint_t power_soft_intr(caddr_t);
145 static uint_t power_issue_shutdown(caddr_t);
146 static void power_timeout(caddr_t);
147 static void power_log_message(void);

149 /*
150  * Structure used in the driver
151  */
152 struct power_soft_state {
153         dev_info_t      *dip;           /* device info pointer */
154         kmutex_t        power_mutex;    /* mutex lock */
155         kmutex_t        power_intr_mutex; /* interrupt mutex lock */
156         ddi_iblock_cookie_t soft_iblock_cookie; /* holds interrupt cookie */
157         ddi_iblock_cookie_t high_iblock_cookie; /* holds interrupt cookie */
158         ddi_softintr_t  softintr_id;    /* soft interrupt id */
159         uchar_t         clones[POWER_MAX_CLONE]; /* array of minor clones */
160         int             monitor_on;     /* clone monitoring the button event */
161                                         /* clone 0 indicates no one is */
162                                         /* monitoring the button event */
163         pollhead_t      pollhd;         /* poll head struct */
164         int             events;         /* bit map of occured events */
165         int             shutdown_pending; /* system shutdown in progress */
166 #ifdef  ACPI_POWER_BUTTON
167         boolean_t       fixed_attached; /* true means fixed is attached */
168         boolean_t       gpe_attached;   /* true means GPE is attached */
169         ACPI_HANDLE     button_obj;     /* handle to device power button */
170 #else
171         ddi_acc_handle_t power_rhandle; /* power button register handle */
172         uint8_t         *power_btn_reg; /* power button register address */
173         uint8_t         power_btn_bit;  /* power button register bit */
174         boolean_t       power_regs_mapped; /* flag to tell if regs mapped */
175         boolean_t       power_btn_ioctl; /* flag to specify ioctl request */
176 #endif
177 };
```

_____***unchanged_portion_omitted_***

****
29723 Mon Apr 28 16:23:09 2014
new/usr/src/uts/sun4u/io/todds1287.c
4823 don't open-code NSEC2MSEC and MSEC2NSEC
****
_____***unchanged_portion_omitted_***

```
 155 static void     *ds1287_state;
 156 static int      instance = -1;

 158 /* Driver Tunables */
 159 static int      ds1287_interrupt_priority = 15;
 160 static int      ds1287_softint_priority = 2;
 161 static hrtime_t power_button_debounce = MSEC2NSEC(10);
 161 static hrtime_t power_button_debounce = NANOSEC/MILLISEC*10;
 162 static hrtime_t power_button_abort_interval = 1.5 * NANOSEC;
 163 static int      power_button_abort_presses = 3;
 164 static int      power_button_abort_enable = 1;
 165 static int      power_button_enable = 1;

 167 static int      power_button_pressed = 0;
 168 static int      power_button_cancel = 0;
 169 static int      power_button_timeouts = 0;
 170 static int      timeout_cancel = 0;
 171 static int      additional_presses = 0;

 173 static ddi_iblock_cookie_t ds1287_lo_iblock;
 174 static ddi_iblock_cookie_t ds1287_hi_iblock;
 175 static ddi_softintr_t   ds1287_softintr_id;
 176 static kmutex_t ds1287_reg_mutex;       /* Protects ds1287 Registers */

 178 static struct modldrv modldrv = {
 179         &mod_driverops,         /* Type of module. This one is a driver */
 180         "ds1287 clock driver",  /* Name of the module. */
 181         &ds1287_ops,            /* driver ops */
 182 };
```

_____***unchanged_portion_omitted_***