```
**********************************************************
    7280 Sun Feb 16 09:52:03 2014
new/usr/src/lib/libdhcpagent/common/dhcp_stable.c
4586 dhcpv6 client id malformed
**********************************************************
```

```
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

/*
 * This module reads and writes the stable identifier values, DUID and IAID.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <fcntl.h>
#include <errno.h>
#include <libdlpi.h>
#include <uuid/uuid.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <net/if.h>
#include <netinet/dhcp6.h>
#include <dhcp_inittab.h>
#include <sys/ethernet.h>
#endif /* ! codereview */

#define DUID_FILE       "/etc/dhcp/duid"
#define IAID_FILE       "/etc/dhcp/iaid"

struct iaid_ent {
        uint32_t        ie_iaid;
        char            ie_name[LIFNAMSIZ];
};

/*
 * read_stable_duid(): read the system's stable DUID, if any
 *
 *   input: size_t *: pointer to a size_t to return the DUID length
```



```
 *  output: uchar_t *: the DUID buffer, or NULL on error (and errno is set)
 *    note: memory returned is from malloc; caller must free.
 */

uchar_t *
read_stable_duid(size_t *duidlen)
{
        int fd;
        ssize_t retv;
        struct stat sb;
        uchar_t *duid = NULL;

        if ((fd = open(DUID_FILE, O_RDONLY)) == -1)
                return (NULL);
        if (fstat(fd, &sb) != -1 && S_ISREG(sb.st_mode) &&
            (duid = malloc(sb.st_size)) != NULL) {
                retv = read(fd, duid, sb.st_size);
                if (retv == sb.st_size) {
                        *duidlen = sb.st_size;
                } else {
                        free(duid);
                        /*
                         * Make sure that errno always gets set when something
                         * goes wrong.
                         */
                        if (retv >= 0)
                                errno = EINVAL;
                        duid = NULL;
                }
        }
        (void) close(fd);
        return (duid);
}

/*
 * write_stable_duid(): write the system's stable DUID.
 *
 *   input: const uchar_t *: pointer to the DUID buffer
 *          size_t: length of the DUID
 *  output: int: 0 on success, -1 on error.  errno is set on error.
 */

int
write_stable_duid(const uchar_t *duid, size_t duidlen)
{
        int fd;
        ssize_t retv;

        (void) unlink(DUID_FILE);
        if ((fd = open(DUID_FILE, O_WRONLY | O_CREAT, 0644)) == -1)
                return (-1);
        retv = write(fd, duid, duidlen);
        if (retv == duidlen) {
                return (close(fd));
        } else {
                (void) close(fd);
                if (retv >= 0)
                        errno = ENOSPC;
                return (-1);
        }
}

/*
 * make_stable_duid(): create a new DUID
 *
 *   input: const char *: name of physical interface for reference
```

```
126  *          size_t *: pointer to a size_t to return the DUID length
127  *  output: uchar_t *: the DUID buffer, or NULL on error (and errno is set)
128  *    note: memory returned is from malloc; caller must free.
129  */

131 uchar_t *
132 make_stable_duid(const char *physintf, size_t *duidlen)
133 {
134         int len;
135         dlpi_info_t dlinfo;
136         dlpi_handle_t dh = NULL;
137         uint_t arptype;
138         duid_en_t *den;

140         /*
141          * Try to read the MAC layer address for the physical interface
142          * provided as a hint.  If that works, we can use a DUID-LLT.
143          */

145         if (dlpi_open(physintf, &dh, 0) == DLPI_SUCCESS &&
146             dlpi_bind(dh, ETHERTYPE_IPV6, NULL) == DLPI_SUCCESS &&
147 #endif /* ! codereview */
148             dlpi_info(dh, &dlinfo, 0) == DLPI_SUCCESS &&
149             (len = dlinfo.di_physaddrlen) > 0 &&
150             (arptype = dlpi_arptype(dlinfo.di_mactype) != 0)) {
151                 duid_llt_t *dllt;
152                 time_t now;

154                 if ((dllt = malloc(sizeof (*dllt) + len)) == NULL) {
155                         dlpi_close(dh);
156                         return (NULL);
157                 }

159                 (void) memcpy((dllt + 1), dlinfo.di_physaddr, len);
160                 dllt->dllt_dutype = htons(DHCPV6_DUID_LLT);
161                 dllt->dllt_hwtype = htons(arptype);
162                 now = time(NULL) - DUID_TIME_BASE;
163                 dllt->dllt_time = htonl(now);
164                 *duidlen = sizeof (*dllt) + len;
165                 dlpi_close(dh);
166                 return ((uchar_t *)dllt);
167         }
168         if (dh != NULL)
169                 dlpi_close(dh);

171         /*
172          * If we weren't able to create a DUID based on the network interface
173          * in use, then generate one based on a UUID.
174          */
175         den = malloc(sizeof (*den) + UUID_LEN);
176         if (den != NULL) {
177                 uuid_t uuid;

179                 den->den_dutype = htons(DHCPV6_DUID_EN);
180                 DHCPV6_SET_ENTNUM(den, DHCPV6_SUN_ENT);
181                 uuid_generate(uuid);
182                 (void) memcpy(den + 1, uuid, UUID_LEN);
183                 *duidlen = sizeof (*den) + UUID_LEN;
184         }
185         return ((uchar_t *)den);
186 }

188 /*
189  * read_stable_iaid(): read a link's stable IAID, if any
190  *
191  *   input: const char *: interface name
```



```
192  *  output: uint32_t: the IAID, or 0 if none
193  */

195 uint32_t
196 read_stable_iaid(const char *intf)
197 {
198         int fd;
199         struct iaid_ent ie;

201         if ((fd = open(IAID_FILE, O_RDONLY)) == -1)
202                 return (0);
203         while (read(fd, &ie, sizeof (ie)) == sizeof (ie)) {
204                 if (strcmp(intf, ie.ie_name) == 0) {
205                         (void) close(fd);
206                         return (ie.ie_iaid);
207                 }
208         }
209         (void) close(fd);
210         return (0);
211 }

213 /*
214  * write_stable_iaid(): write out a link's stable IAID
215  *
216  *   input: const char *: interface name
217  *  output: uint32_t: the IAID, or 0 if none
218  */

220 int
221 write_stable_iaid(const char *intf, uint32_t iaid)
222 {
223         int fd;
224         struct iaid_ent ie;
225         ssize_t retv;

227         if ((fd = open(IAID_FILE, O_RDWR | O_CREAT, 0644)) == -1)
228                 return (0);
229         while (read(fd, &ie, sizeof (ie)) == sizeof (ie)) {
230                 if (strcmp(intf, ie.ie_name) == 0) {
231                         (void) close(fd);
232                         if (iaid == ie.ie_iaid) {
233                                 return (0);
234                         } else {
235                                 errno = EINVAL;
236                                 return (-1);
237                         }
238                 }
239         }
240         (void) memset(&ie, 0, sizeof (ie));
241         ie.ie_iaid = iaid;
242         (void) strlcpy(ie.ie_name, intf, sizeof (ie.ie_name));
243         retv = write(fd, &ie, sizeof (ie));
244         (void) close(fd);
245         if (retv == sizeof (ie)) {
246                 return (0);
247         } else {
248                 if (retv >= 0)
249                         errno = ENOSPC;
250                 return (-1);
251         }
252 }

254 /*
255  * make_stable_iaid(): create a stable IAID for a link
256  *
257  *   input: const char *: interface name
```

```
 *          uint32_t: the ifIndex for this link (as a "hint")
 *  output: uint32_t: the new IAID, never zero
 */

/* ARGSUSED */
uint32_t
make_stable_iaid(const char *intf, uint32_t hint)
{
        int fd;
        struct iaid_ent ie;
        uint32_t maxid, minunused;
        boolean_t recheck;

        if ((fd = open(IAID_FILE, O_RDONLY)) == -1)
                return (hint);
        maxid = 0;
        minunused = 1;
        /*
         * This logic is deliberately unoptimized.  The reason is that it runs
         * essentially just once per interface for the life of the system.
         * Once the IAID is established, there's no reason to generate it
         * again, and all we care about here is correctness.  Also, IAIDs tend
         * to get added in a logical sequence order, so the outer loop should
         * not normally run more than twice.
         */
        do {
                recheck = B_FALSE;
                while (read(fd, &ie, sizeof (ie)) == sizeof (ie)) {
                        if (ie.ie_iaid > maxid)
                                maxid = ie.ie_iaid;
                        if (ie.ie_iaid == minunused) {
                                recheck = B_TRUE;
                                minunused++;
                        }
                        if (ie.ie_iaid == hint)
                                hint = 0;
                }
                if (recheck)
                        (void) lseek(fd, 0, SEEK_SET);
        } while (recheck);
        (void) close(fd);
        if (hint != 0)
                return (hint);
        else if (maxid != UINT32_MAX)
                return (maxid + 1);
        else
                return (minunused);
}
```

```
**********************************************************
   47819 Sun Feb 16 09:52:04 2014
new/usr/src/lib/libdhcputil/common/dhcp_inittab.c
4586 dhcpv6 client id malformed
**********************************************************
```

```
/*
 * 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.
 */

#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdarg.h>
#include <limits.h>
#include <ctype.h>
#include <libgen.h>
#include <sys/isa_defs.h>
#include <sys/socket.h>
#include <net/if_arp.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/sysmacros.h>
#include <libinetutil.h>
#include <libdlpi.h>
#include <netinet/dhcp6.h>
#include <sys/ethernet.h>
#endif /* ! codereview */

#include "dhcp_symbol.h"
#include "dhcp_inittab.h"

static void             inittab_msg(const char *, ...);
static uchar_t          category_to_code(const char *);
static boolean_t        encode_number(uint8_t, uint8_t, boolean_t, uint8_t,
                            const char *, uint8_t *, int *);
static boolean_t        decode_number(uint8_t, uint8_t, boolean_t, uint8_t,
                            const uint8_t *, char *, int *);
static dhcp_symbol_t    *inittab_lookup(uchar_t, char, const char *, int32_t,
                            size_t *);
static dsym_category_t  itabcode_to_dsymcode(uchar_t);
static boolean_t        parse_entry(char *, char **);

/*
```



```
 * forward declaration of our internal inittab_table[].  too bulky to put
 * up front -- check the end of this file for its definition.
 *
 * Note: we have only an IPv4 version here.  The inittab_verify() function is
 * used by the DHCP server and manager.  We'll need a new function if the
 * server is extended to DHCPv6.
 */
static dhcp_symbol_t    inittab_table[];

/*
 * the number of fields in the inittab and names for the fields.  note that
 * this order is meaningful to parse_entry(); other functions should just
 * use them as indexes into the array returned from parse_entry().
 */
#define ITAB_FIELDS     7
enum { ITAB_NAME, ITAB_CODE, ITAB_TYPE, ITAB_GRAN, ITAB_MAX, ITAB_CONS,
    ITAB_CAT };

/*
 * the category_map_entry_t is used to map the inittab category codes to
 * the dsym codes.  the reason the codes are different is that the inittab
 * needs to have the codes be ORable such that queries can retrieve more
 * than one category at a time.  this map is also used to map the inittab
 * string representation of a category to its numerical code.
 */
typedef struct category_map_entry {
        dsym_category_t cme_dsymcode;
        char            *cme_name;
        uchar_t         cme_itabcode;
} category_map_entry_t;

static category_map_entry_t category_map[] = {
        { DSYM_STANDARD,        "STANDARD",     ITAB_CAT_STANDARD },
        { DSYM_FIELD,           "FIELD",        ITAB_CAT_FIELD },
        { DSYM_INTERNAL,        "INTERNAL",     ITAB_CAT_INTERNAL },
        { DSYM_VENDOR,          "VENDOR",       ITAB_CAT_VENDOR },
        { DSYM_SITE,            "SITE",         ITAB_CAT_SITE }
};

/*
 * inittab_load(): returns all inittab entries with the specified criteria
 *
 *   input: uchar_t: the categories the consumer is interested in
 *          char: the consumer type of the caller
 *          size_t *: set to the number of entries returned
 *  output: dhcp_symbol_t *: an array of dynamically allocated entries
 *          on success, NULL upon failure
 */

dhcp_symbol_t   *
inittab_load(uchar_t categories, char consumer, size_t *n_entries)
{
        return (inittab_lookup(categories, consumer, NULL, -1, n_entries));
}

/*
 * inittab_getbyname(): returns an inittab entry with the specified criteria
 *
 *   input: int: the categories the consumer is interested in
 *          char: the consumer type of the caller
 *          char *: the name of the inittab entry the consumer wants
 *  output: dhcp_symbol_t *: a dynamically allocated dhcp_symbol structure
 *          on success, NULL upon failure
 */

dhcp_symbol_t   *
```

```
128 inittab_getbyname(uchar_t categories, char consumer, const char *name)
129 {
130         return (inittab_lookup(categories, consumer, name, -1, NULL));
131 }

133 /*
134  * inittab_getbycode(): returns an inittab entry with the specified criteria
135  *
136  *   input: uchar_t: the categories the consumer is interested in
137  *          char: the consumer type of the caller
138  *          uint16_t: the code of the inittab entry the consumer wants
139  *  output: dhcp_symbol_t *: a dynamically allocated dhcp_symbol structure
140  *          on success, NULL upon failure
141  */

143 dhcp_symbol_t   *
144 inittab_getbycode(uchar_t categories, char consumer, uint16_t code)
145 {
146         return (inittab_lookup(categories, consumer, NULL, code, NULL));
147 }

149 /*
150  * inittab_lookup(): returns inittab entries with the specified criteria
151  *
152  *   input: uchar_t: the categories the consumer is interested in
153  *          char: the consumer type of the caller
154  *          const char *: the name of the entry the caller is interested
155  *              in, or NULL if the caller doesn't care
156  *          int32_t: the code the caller is interested in, or -1 if the
157  *              caller doesn't care
158  *          size_t *: set to the number of entries returned
159  *  output: dhcp_symbol_t *: dynamically allocated dhcp_symbol structures
160  *          on success, NULL upon failure
161  */

163 static dhcp_symbol_t *
164 inittab_lookup(uchar_t categories, char consumer, const char *name,
165     int32_t code, size_t *n_entriesp)
166 {
167         FILE                    *inittab_fp;
168         dhcp_symbol_t           *new_entries, *entries = NULL;
169         dhcp_symbol_t           entry;
170         char                    buffer[ITAB_MAX_LINE_LEN];
171         char                    *fields[ITAB_FIELDS];
172         unsigned long           line = 0;
173         size_t                  i, n_entries = 0;
174         const char              *inittab_path;
175         uchar_t                 category_code;
176         dsym_cdtype_t           type;

178         if (categories & ITAB_CAT_V6) {
179                 inittab_path = getenv("DHCP_INITTAB6_PATH");
180                 if (inittab_path == NULL)
181                         inittab_path = ITAB_INITTAB6_PATH;
182         } else {
183                 inittab_path = getenv("DHCP_INITTAB_PATH");
184                 if (inittab_path == NULL)
185                         inittab_path = ITAB_INITTAB_PATH;
186         }

188         inittab_fp = fopen(inittab_path, "r");
189         if (inittab_fp == NULL) {
190                 inittab_msg("inittab_lookup: fopen: %s: %s",
191                     inittab_path, strerror(errno));
192                 return (NULL);
193         }
```



```
195         (void) bufsplit(",\n", 0, NULL);
196         while (fgets(buffer, sizeof (buffer), inittab_fp) != NULL) {

198                 line++;

200                 /*
201                  * make sure the string didn't overflow our buffer
202                  */
203                 if (strchr(buffer, '\n') == NULL) {
204                         inittab_msg("inittab_lookup: line %li: too long, "
205                             "skipping", line);
206                         continue;
207                 }

209                 /*
210                  * skip 'pure comment' lines
211                  */
212                 for (i = 0; buffer[i] != '\0'; i++)
213                         if (isspace(buffer[i]) == 0)
214                                 break;

216                 if (buffer[i] == ITAB_COMMENT_CHAR || buffer[i] == '\0')
217                         continue;

219                 /*
220                  * parse the entry out into fields.
221                  */
222                 if (parse_entry(buffer, fields) == B_FALSE) {
223                         inittab_msg("inittab_lookup: line %li: syntax error, "
224                             "skipping", line);
225                         continue;
226                 }

228                 /*
229                  * validate the values in the entries; skip if invalid.
230                  */
231                 if (atoi(fields[ITAB_GRAN]) > ITAB_GRAN_MAX) {
232                         inittab_msg("inittab_lookup: line %li: granularity '%s'"
233                             " out of range, skipping", line, fields[ITAB_GRAN]);
234                         continue;
235                 }

237                 if (atoi(fields[ITAB_MAX]) > ITAB_MAX_MAX) {
238                         inittab_msg("inittab_lookup: line %li: maximum '%s' "
239                             "out of range, skipping", line, fields[ITAB_MAX]);
240                         continue;
241                 }

243                 if (dsym_get_type_id(fields[ITAB_TYPE], &type, B_FALSE) !=
244                     DSYM_SUCCESS) {
245                         inittab_msg("inittab_lookup: line %li: type '%s' "
246                             "is invalid, skipping", line, fields[ITAB_TYPE]);
247                         continue;
248                 }

250                 /*
251                  * find out whether this entry of interest to our consumer,
252                  * and if so, throw it onto the set of entries we'll return.
253                  * check categories last since it's the most expensive check.
254                  */
255                 if (strchr(fields[ITAB_CONS], consumer) == NULL)
256                         continue;

258                 if (code != -1 && atoi(fields[ITAB_CODE]) != code)
259                         continue;
```

```
261                 if (name != NULL && strcasecmp(fields[ITAB_NAME], name) != 0)
262                         continue;

264                 category_code = category_to_code(fields[ITAB_CAT]);
265                 if ((category_code & categories) == 0)
266                         continue;

268                 /*
269                  * looks like a match.  allocate an entry and fill it in
270                  */
271                 new_entries = realloc(entries, (n_entries + 1) *
272                     sizeof (dhcp_symbol_t));

274                 /*
275                  * if we run out of memory, might as well return what we can
276                  */
277                 if (new_entries == NULL) {
278                         inittab_msg("inittab_lookup: ran out of memory "
279                             "allocating dhcp_symbol_t's");
280                         break;
281                 }

283                 entry.ds_max      = atoi(fields[ITAB_MAX]);
284                 entry.ds_code     = atoi(fields[ITAB_CODE]);
285                 entry.ds_type     = type;
286                 entry.ds_gran     = atoi(fields[ITAB_GRAN]);
287                 entry.ds_category = itabcode_to_dsymcode(category_code);
288                 entry.ds_classes.dc_cnt   = 0;
289                 entry.ds_classes.dc_names = NULL;
290                 (void) strlcpy(entry.ds_name, fields[ITAB_NAME],
291                     sizeof (entry.ds_name));
292                 entry.ds_dhcpv6   = (categories & ITAB_CAT_V6) ? 1 : 0;

294                 entries = new_entries;
295                 entries[n_entries++] = entry;
296         }

298         if (ferror(inittab_fp) != 0) {
299                 inittab_msg("inittab_lookup: error on inittab stream");
300                 clearerr(inittab_fp);
301         }

303         (void) fclose(inittab_fp);

305         if (n_entriesp != NULL)
306                 *n_entriesp = n_entries;

308         return (entries);
309 }

311 /*
312  * parse_entry(): parses an entry out into its constituent fields
313  *
314  *   input: char *: the entry
315  *          char **: an array of ITAB_FIELDS length which contains
316  *                   pointers into the entry on upon return
317  *  output: boolean_t: B_TRUE on success, B_FALSE on failure
318  */

320 static boolean_t
321 parse_entry(char *entry, char **fields)
322 {
323         char    *category, *spacep;
324         size_t  n_fields, i;
```



```
326         /*
327          * due to a mistake made long ago, the first and second fields of
328          * each entry are not separated by a comma, but rather by
329          * whitespace -- have bufsplit() treat the two fields as one, then
330          * pull them apart afterwards.
331          */
332         n_fields = bufsplit(entry, ITAB_FIELDS - 1, fields);
333         if (n_fields != (ITAB_FIELDS - 1))
334                 return (B_FALSE);

336         /*
337          * pull the first and second fields apart.  this is complicated
338          * since the first field can contain embedded whitespace (so we
339          * must separate the two fields by the last span of whitespace).
340          *
341          * first, find the initial span of whitespace.  if there isn't one,
342          * then the entry is malformed.
343          */
344         category = strpbrk(fields[ITAB_NAME], " \t");
345         if (category == NULL)
346                 return (B_FALSE);

348         /*
349          * find the last span of whitespace.
350          */
351         do {
352                 while (isspace(*category))
353                         category++;

355                 spacep = strpbrk(category, " \t");
356                 if (spacep != NULL)
357                         category = spacep;
358         } while (spacep != NULL);

360         /*
361          * NUL-terminate the first byte of the last span of whitespace, so
362          * that the first field doesn't have any residual trailing
363          * whitespace.
364          */
365         spacep = category - 1;
366         while (isspace(*spacep))
367                 spacep--;

369         if (spacep <= fields[0])
370                 return (B_FALSE);

372         *++spacep = '\0';

374         /*
375          * remove any whitespace from the fields.
376          */
377         for (i = 0; i < n_fields; i++) {
378                 while (isspace(*fields[i]))
379                         fields[i]++;
380         }
381         fields[ITAB_CAT] = category;

383         return (B_TRUE);
384 }

386 /*
387  * inittab_verify(): verifies that a given inittab entry matches an internal
388  *                   definition
389  *
390  *   input: dhcp_symbol_t *: the inittab entry to verify
391  *          dhcp_symbol_t *: if non-NULL, a place to store the internal
```

```
 *                              inittab entry upon return
 *  output: int: ITAB_FAILURE, ITAB_SUCCESS, or ITAB_UNKNOWN
 *
 *   notes: IPv4 only
 */

int
inittab_verify(const dhcp_symbol_t *inittab_ent, dhcp_symbol_t *internal_ent)
{
        unsigned int    i;

        for (i = 0; inittab_table[i].ds_name[0] != '\0'; i++) {

                if (inittab_ent->ds_category != inittab_table[i].ds_category)
                        continue;

                if (inittab_ent->ds_code == inittab_table[i].ds_code) {
                        if (internal_ent != NULL)
                                *internal_ent = inittab_table[i];

                        if (inittab_table[i].ds_type != inittab_ent->ds_type ||
                            inittab_table[i].ds_gran != inittab_ent->ds_gran ||
                            inittab_table[i].ds_max  != inittab_ent->ds_max)
                                return (ITAB_FAILURE);

                        return (ITAB_SUCCESS);
                }
        }

        return (ITAB_UNKNOWN);
}

/*
 * get_hw_type(): interpret ",hwtype" in the input string, as part of a DUID.
 *                The hwtype string is optional, and must be 0-65535 if
 *                present.
 *
 *   input: char **: pointer to string pointer
 *          int *: error return value
 *  output: int: hardware type, or -1 for empty, or -2 for error.
 */

static int
get_hw_type(char **strp, int *ierrnop)
{
        char *str = *strp;
        ulong_t hwtype;

        if (*str++ != ',') {
                *ierrnop = ITAB_BAD_NUMBER;
                return (-2);
        }
        if (*str == ',' || *str == '\0') {
                *strp = str;
                return (-1);
        }
        hwtype = strtoul(str, strp, 0);
        if (errno != 0 || *strp == str || hwtype > 65535) {
                *ierrnop = ITAB_BAD_NUMBER;
                return (-2);
        } else {
                return ((int)hwtype);
        }
}

/*
```



```
 * get_mac_addr(): interpret ",macaddr" in the input string, as part of a DUID.
 *                 The 'macaddr' may be a hex string (in any standard format),
 *                 or the name of a physical interface.  If an interface name
 *                 is given, then the interface type is extracted as well.
 *
 *   input: const char *: input string
 *          int *: error return value
 *          uint16_t *: hardware type output (network byte order)
 *          int: hardware type input; -1 for empty
 *          uchar_t *: output buffer for MAC address
 *  output: int: length of MAC address, or -1 for error
 */

static int
get_mac_addr(const char *str, int *ierrnop, uint16_t *hwret, int hwtype,
    uchar_t *outbuf)
{
        int maclen;
        int dig, val;
        dlpi_handle_t dh;
        dlpi_info_t dlinfo;
        char chr;

        if (*str != '\0') {
                if (*str++ != ',')
                        goto failed;
                if (dlpi_open(str, &dh, 0) != DLPI_SUCCESS) {
                        maclen = 0;
                        dig = val = 0;
                        /*
                         * Allow MAC addresses with separators matching regexp
                         * (:|-| *).
                         */
                        while ((chr = *str++) != '\0') {
                                if (isdigit(chr)) {
                                        val = (val << 4) + chr - '0';
                                } else if (isxdigit(chr)) {
                                        val = (val << 4) + chr -
                                            (isupper(chr) ? 'A' : 'a') + 10;
                                } else if (isspace(chr) && dig == 0) {
                                        continue;
                                } else if (chr == ':' || chr == '-' ||
                                    isspace(chr)) {
                                        dig = 1;
                                } else {
                                        goto failed;
                                }
                                if (++dig == 2) {
                                        *outbuf++ = val;
                                        maclen++;
                                        dig = val = 0;
                                }
                        }
                } else {
                        if (dlpi_bind(dh, ETHERTYPE_IPV6, NULL) !=
                            DLPI_SUCCESS || dlpi_info(dh, &dlinfo, 0) !=
                            DLPI_SUCCESS) {
                        if (dlpi_info(dh, &dlinfo, 0) != DLPI_SUCCESS) {
                                dlpi_close(dh);
                                goto failed;
                        }
                        maclen = dlinfo.di_physaddrlen;
                        (void) memcpy(outbuf, dlinfo.di_physaddr, maclen);
                        dlpi_close(dh);
                        if (hwtype == -1)
                                hwtype = dlpi_arptype(dlinfo.di_mactype);
```

```
 523                 }
 524         }
 525         if (hwtype == -1)
 526                 goto failed;
 527         *hwret = htons(hwtype);
 528         return (maclen);

 530 failed:
 531         *ierrnop = ITAB_BAD_NUMBER;
 532         return (-1);
 533 }
```

_____***unchanged_portion_omitted_***