```
**********************************************************
    1760 Sun Feb  9 05:30:58 2014
new/usr/src/cmd/dladm/Makefile
4585 dladm(1m) needs a 'help' subcommand
3755 dladm show-aggr documentation
3374 usage of 'dladm' does not match to its man page
**********************************************************
```

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

  26 PROG= dladm
  27 CFGFILES= secobj.conf

  29 ROOTFS_PROG=    $(PROG)
  30 ROOTCFGDIR=     $(ROOTETC)/dladm
  31 ROOTCFGFILES=   $(CFGFILES:%=$(ROOTCFGDIR)/%)

  33 include ../Makefile.cmd

  35 XGETFLAGS += -a -x $(PROG).xcl
  36 LDLIBS += -L$(ROOT)/lib -lsocket
  37 LDLIBS += -ldladm -ldlpi -lkstat -lsecdb -lbsm -linetutil -ldevinfo
  38 LDLIBS += $(ZLAZYLOAD) -lrstp $(ZNOLAZYLOAD)

  40 CERRWARN += -_gcc=-Wno-switch
  41 CERRWARN += -_gcc=-Wno-unused-label
  41 CERRWARN += -_gcc=-Wno-uninitialized

  43 # For headers from librstp.
  44 LINTFLAGS +=    -erroff=E_TRAILING_COMMA_IN_ENUM

  46 $(ROOTCFGDIR)/secobj.conf := FILEMODE= 660

  48 lint := ZLAZYLOAD=
  49 lint := ZNOLAZYLOAD=

  51 .KEEP_STATE:

  53 all: $(ROOTFS_PROG)

  55 install: all $(ROOTSBINPROG) $(ROOTCFGDIR) $(ROOTCFGFILES)
  56         $(RM) $(ROOTUSRSBINPROG)
  57         -$(SYMLINK) ../../sbin/$(PROG) $(ROOTUSRSBINPROG)
```



```
  59 clean:

  61 lint:   lint_PROG

  63 $(ROOTCFGDIR):
  64         $(INS.dir)

  66 $(ROOTCFGDIR)/%: $(ROOTCFGDIR) %
  67         $(INS.file)

  69 include ../Makefile.targ
```

```
**********************************************************
  255995 Sun Feb  9 05:30:58 2014
new/usr/src/cmd/dladm/dladm.c
4585 dladm(1m) needs a 'help' subcommand
3755 dladm show-aggr documentation
3374 usage of 'dladm' does not match to its man page
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */

  25 #include <stdio.h>
  26 #include <ctype.h>
  27 #include <dlfcn.h>
  28 #include <locale.h>
  29 #include <signal.h>
  30 #include <stdarg.h>
  31 #include <stdlib.h>
  32 #include <fcntl.h>
  33 #include <string.h>
  34 #include <stropts.h>
  35 #include <sys/stat.h>
  36 #include <errno.h>
  37 #include <kstat.h>
  38 #include <strings.h>
  39 #include <getopt.h>
  40 #include <unistd.h>
  41 #include <priv.h>
  42 #include <limits.h>
  43 #include <termios.h>
  44 #include <pwd.h>
  45 #include <auth_attr.h>
  46 #include <auth_list.h>
  47 #include <libintl.h>
  48 #include <libdevinfo.h>
  49 #include <libdlpi.h>
  50 #include <libdladm.h>
  51 #include <libdllink.h>
  52 #include <libdlstat.h>
  53 #include <libdlaggr.h>
  54 #include <libdlwlan.h>
  55 #include <libdlvlan.h>
  56 #include <libdlvnic.h>
  57 #include <libdlib.h>
  58 #include <libdlether.h>
  59 #include <libdliptun.h>
```



```
  60 #include <libdlsim.h>
  61 #include <libdlbridge.h>
  62 #include <libinetutil.h>
  63 #include <libvrrpadm.h>
  64 #include <bsm/adt.h>
  65 #include <bsm/adt_event.h>
  66 #include <libdlvnic.h>
  67 #include <sys/types.h>
  68 #include <sys/socket.h>
  69 #include <sys/ib/ib_types.h>
  70 #include <sys/processor.h>
  71 #include <netinet/in.h>
  72 #include <arpa/inet.h>
  73 #include <net/if_types.h>
  74 #include <stddef.h>
  75 #include <stp_in.h>
  76 #include <ofmt.h>

  78 #define MAXPORT                 256
  79 #define MAXVNIC                 256
  80 #define BUFLEN(lim, ptr)        (((lim) > (ptr)) ? ((lim) - (ptr)) : 0)
  81 #define MAXLINELEN              1024
  82 #define SMF_UPGRADE_FILE                "/var/svc/profile/upgrade"
  83 #define SMF_UPGRADEDATALINK_FILE        "/var/svc/profile/upgrade_datalink"
  84 #define SMF_DLADM_UPGRADE_MSG           " # added by dladm(1M)"
  85 #define DLADM_DEFAULT_COL       80
  86 #define DLADM_DEFAULT_CMD       "show-link"
  87 #endif /* ! codereview */

  89 /*
  90  * used by the wifi show-* commands to set up ofmt_field_t structures.
  91  */
  92 #define WIFI_CMD_SCAN           0x00000001
  93 #define WIFI_CMD_SHOW           0x00000002
  94 #define WIFI_CMD_ALL            (WIFI_CMD_SCAN | WIFI_CMD_SHOW)

  96 /* No larger than pktsum_t */
  97 typedef struct brsum_s {
  98         uint64_t        drops;
  99         uint64_t        forward_dir;
 100         uint64_t        forward_mb;
 101         uint64_t        forward_unk;
 102         uint64_t        recv;
 103         uint64_t        sent;
 104 } brsum_t;

 106 /* No larger than pktsum_t */
 107 typedef struct brlsum_s {
 108         uint32_t        cfgbpdu;
 109         uint32_t        tcnbpdu;
 110         uint32_t        rstpbpdu;
 111         uint32_t        txbpdu;
 112         uint64_t        drops;
 113         uint64_t        recv;
 114         uint64_t        xmit;
 115 } brlsum_t;

 117 typedef struct show_state {
 118         boolean_t       ls_firstonly;
 119         boolean_t       ls_donefirst;
 120         pktsum_t        ls_prevstats;
 121         uint32_t        ls_flags;
 122         dladm_status_t  ls_status;
 123         ofmt_handle_t   ls_ofmt;
 124         boolean_t       ls_parsable;
 125         boolean_t       ls_mac;
```

```
126         boolean_t       ls_hwgrp;
127 } show_state_t;

129 typedef struct show_grp_state {
130         pktsum_t        gs_prevstats[MAXPORT];
131         uint32_t        gs_flags;
132         dladm_status_t  gs_status;
133         boolean_t       gs_parsable;
134         boolean_t       gs_lacp;
135         boolean_t       gs_extended;
136         boolean_t       gs_stats;
137         boolean_t       gs_firstonly;
138         boolean_t       gs_donefirst;
139         ofmt_handle_t   gs_ofmt;
140 } show_grp_state_t;

142 typedef struct show_vnic_state {
143         datalink_id_t   vs_vnic_id;
144         datalink_id_t   vs_link_id;
145         char            vs_vnic[MAXLINKNAMELEN];
146         char            vs_link[MAXLINKNAMELEN];
147         boolean_t       vs_parsable;
148         boolean_t       vs_found;
149         boolean_t       vs_firstonly;
150         boolean_t       vs_donefirst;
151         boolean_t       vs_stats;
152         boolean_t       vs_printstats;
153         pktsum_t        vs_totalstats;
154         pktsum_t        vs_prevstats[MAXVNIC];
155         boolean_t       vs_etherstub;
156         dladm_status_t  vs_status;
157         uint32_t        vs_flags;
158         ofmt_handle_t   vs_ofmt;
159 } show_vnic_state_t;

161 typedef struct show_part_state {
162         datalink_id_t   ps_over_id;
163         char            ps_part[MAXLINKNAMELEN];
164         boolean_t       ps_parsable;
165         boolean_t       ps_found;
166         dladm_status_t  ps_status;
167         uint32_t        ps_flags;
168         ofmt_handle_t   ps_ofmt;
169 } show_part_state_t;

171 typedef struct show_ib_state {
172         datalink_id_t   is_link_id;
173         char            is_link[MAXLINKNAMELEN];
174         boolean_t       is_parsable;
175         dladm_status_t  is_status;
176         uint32_t        is_flags;
177         ofmt_handle_t   is_ofmt;
178 } show_ib_state_t;

180 typedef struct show_usage_state_s {
181         boolean_t       us_plot;
182         boolean_t       us_parsable;
183         boolean_t       us_printheader;
184         boolean_t       us_first;
185         boolean_t       us_showall;
186         ofmt_handle_t   us_ofmt;
187 } show_usage_state_t;

189 /*
190  * callback functions for printing output and error diagnostics.
191  */
```



```
192 static ofmt_cb_t print_default_cb, print_link_stats_cb, print_linkprop_cb;
193 static ofmt_cb_t print_lacp_cb, print_phys_one_mac_cb;
194 static ofmt_cb_t print_xaggr_cb, print_aggr_stats_cb;
195 static ofmt_cb_t print_phys_one_hwgrp_cb, print_wlan_attr_cb;
196 static ofmt_cb_t print_wifi_status_cb, print_link_attr_cb;
197 static void dladm_ofmt_check(ofmt_status_t, boolean_t, ofmt_handle_t);

199 typedef void cmdfunc_t(int, char **, const char *);

201 static cmdfunc_t do_show_link, do_show_wifi, do_show_phys;
202 static cmdfunc_t do_create_aggr, do_delete_aggr, do_add_aggr, do_remove_aggr;
203 static cmdfunc_t do_modify_aggr, do_show_aggr, do_up_aggr;
204 static cmdfunc_t do_scan_wifi, do_connect_wifi, do_disconnect_wifi;
205 static cmdfunc_t do_show_linkprop, do_set_linkprop, do_reset_linkprop;
206 static cmdfunc_t do_create_secobj, do_delete_secobj, do_show_secobj;
207 static cmdfunc_t do_init_linkprop, do_init_secobj;
208 static cmdfunc_t do_create_vlan, do_delete_vlan, do_up_vlan, do_show_vlan;
209 static cmdfunc_t do_rename_link, do_delete_phys, do_init_phys;
210 static cmdfunc_t do_show_linkmap;
211 static cmdfunc_t do_show_ether;
212 static cmdfunc_t do_create_vnic, do_delete_vnic, do_show_vnic;
213 static cmdfunc_t do_up_vnic;
214 static cmdfunc_t do_create_part, do_delete_part, do_show_part, do_show_ib;
215 static cmdfunc_t do_up_part;
216 static cmdfunc_t do_create_etherstub, do_delete_etherstub, do_show_etherstub;
217 static cmdfunc_t do_create_simnet, do_modify_simnet;
218 static cmdfunc_t do_delete_simnet, do_show_simnet, do_up_simnet;
219 static cmdfunc_t do_show_usage;
220 static cmdfunc_t do_create_bridge, do_modify_bridge, do_delete_bridge;
221 static cmdfunc_t do_add_bridge, do_remove_bridge, do_show_bridge;
222 static cmdfunc_t do_create_iptun, do_modify_iptun, do_delete_iptun;
223 static cmdfunc_t do_show_iptun, do_up_iptun, do_down_iptun;
224 static cmdfunc_t do_help;
225 #endif /* ! codereview */

227 static void     do_up_vnic_common(int, char **, const char *, boolean_t);

229 static int show_part(dladm_handle_t, datalink_id_t, void *);

231 static void     altroot_cmd(char *, int, char **);
232 static int      show_linkprop_onelink(dladm_handle_t, datalink_id_t, void *);

234 static void     link_stats(datalink_id_t, uint_t, char *, show_state_t *);
235 static void     aggr_stats(datalink_id_t, show_grp_state_t *, uint_t);
236 static void     vnic_stats(show_vnic_state_t *, uint32_t);

238 static int      get_one_kstat(const char *, const char *, uint8_t,
239                     void *, boolean_t);
240 static void     get_mac_stats(const char *, pktsum_t *);
241 static void     get_link_stats(const char *, pktsum_t *);
242 static uint64_t get_ifspeed(const char *, boolean_t);
243 static const char       *get_linkstate(const char *, boolean_t, char *);
244 static const char       *get_linkduplex(const char *, boolean_t, char *);

246 static iptun_type_t     iptun_gettypebyname(char *);
247 static const char       *iptun_gettypebyvalue(iptun_type_t);
248 static dladm_status_t   print_iptun(dladm_handle_t, datalink_id_t,
249                             show_state_t *);
250 static int      print_iptun_walker(dladm_handle_t, datalink_id_t, void *);

252 static int      show_etherprop(dladm_handle_t, datalink_id_t, void *);
253 static void     show_ether_xprop(void *, dladm_ether_info_t *);
254 static boolean_t        link_is_ether(const char *, datalink_id_t *);

256 static boolean_t str2int(const char *, int *);
257 static void     die(const char *, ...);
```

```
258 static void     die_optdup(int);
259 static void     die_opterr(int, int, const char *);
260 static void     die_dlerr(dladm_status_t, const char *, ...);
261 static void     warn(const char *, ...);
262 static void     warn_dlerr(dladm_status_t, const char *, ...);

264 typedef struct  cmd {
265         char            *c_name;
266         cmdfunc_t       *c_fn;
267         const char      *c_usage;
268 } cmd_t;

270 static cmd_t    cmds[] = {
271         { "rename-link",        do_rename_link,
272           "    rename-link      <oldlink> <newlink>"                  },
273         { "show-link",          do_show_link,
274           "    show-link        [-P] [[-p] -o <field>,..] "
275           "[-s [-i <interval>]] [<link>]"                             },
 86           "    show-link        [-pP] [-o <field>,..] [-s [-i <interval>]] "
 87           "[<link>]\n"                                               },
276         { "create-aggr",        do_create_aggr,
277           "    create-aggr      [-t] [-P <policy>] [-L <mode>] [-T <time>] "
278           "[-u <address>]\n"
279           "\t\t     -l <link> [-l <link>...] <link>"                 },
280         { "delete-aggr",        do_delete_aggr,
281           "    delete-aggr      [-t] <link>"                         },
282         { "add-aggr",           do_add_aggr,
283           "    add-aggr         [-t] -l <link> [-l <link>...] <link>" },
284         { "remove-aggr",        do_remove_aggr,
285           "    remove-aggr      [-t] -l <link> [-l <link>...] <link>" },
286         { "modify-aggr",        do_modify_aggr,
287           "    modify-aggr      [-t] [-P <policy>] [-L <mode>] [-T <time>] "
288           "[-u <address>]\n"
289           "\t\t     <link>"                                         },
290         { "show-aggr",          do_show_aggr,
291           "    show-aggr        [-PLx] [[-p] -o <field>,..] "
292           "[-s [-i <interval>]] [<link>]"                             },
103           "    show-aggr        [-pPLx] [-o <field>,..] [-s [-i <interval>]] "
104           "[<link>]\n"                                               },
293         { "up-aggr",            do_up_aggr,     NULL                },
294         { "scan-wifi",          do_scan_wifi,
295           "    scan-wifi        [[-p] -o <field>,...] [<link>]"      },
107           "    scan-wifi        [-p] [-o <field>,...] [<link>]"      },
296         { "connect-wifi",       do_connect_wifi,
297           "    connect-wifi     [-e <essid>] [-i <bssid>] [-k <key>,...] "
298           "[-s wep|wpa]\n"
299           "\t\t     [-a open|shared] [-b bss|ibss] [-c] [-m a|b|g] "
300           "[-T <time>]\n"
301           "\t\t     [<link>]"                                       },
302         { "disconnect-wifi",    do_disconnect_wifi,
303           "    disconnect-wifi  [-a] [<link>]"                      },
304         { "show-wifi",          do_show_wifi,
305           "    show-wifi        [[-p] -o <field>,...] [<link>]"      },
117           "    show-wifi        [-p] [-o <field>,...] [<link>]\n"    },
306         { "set-linkprop",       do_set_linkprop,
307           "    set-linkprop     [-t] -p <prop>=<value>[,...] <name>" },
308         { "reset-linkprop",     do_reset_linkprop,
309           "    reset-linkprop   [-t] [-p <prop>,...] <name>"         },
310         { "show-linkprop",      do_show_linkprop,
311           "    show-linkprop    [-cP] [-o <field>,...] [-p <prop>,...] "
312           "<name>"                                                   },
124           "<name>\n"                                                 },
313         { "show-ether",         do_show_ether,
314           "    show-ether       [-x] [[-p] -o <field>,...] <link>"   },
126           "    show-ether       [-px][-o <field>,...] <link>\n"      },
315         { "create-secobj",      do_create_secobj,
```



```
316           "    create-secobj    [-t] [-f <file>] -c <class> <secobj>" },
317         { "delete-secobj",      do_delete_secobj,
318           "    delete-secobj    [-t] <secobj>[,...]"                 },
319         { "show-secobj",        do_show_secobj,
320           "    show-secobj      [-P] [[-p] -o <field>,...] [<secobj>,...]" },
132           "    show-secobj      [-pP] [-o <field>,...] [<secobj>,...]\n" },
321         { "init-linkprop",      do_init_linkprop,       NULL        },
322         { "init-secobj",        do_init_secobj,         NULL        },
323         { "create-vlan",        do_create_vlan,
324           "    create-vlan      [-ft] -l <link> -v <vid> [link]"     },
325         { "delete-vlan",        do_delete_vlan,
326           "    delete-vlan      [-t] <link>"                         },
327         { "show-vlan",          do_show_vlan,
328           "    show-vlan        [-P] [[-p] -o <field>,...] [<link>]"  },
140           "    show-vlan        [-pP] [-o <field>,..] [<link>]\n"    },
329         { "up-vlan",            do_up_vlan,             NULL        },
330         { "create-iptun",       do_create_iptun,
331           "    create-iptun     [-t] -T <type> "
332           "[-a {local|remote}=<addr>,...] <link>]" },
333         { "delete-iptun",       do_delete_iptun,
334           "    delete-iptun     [-t] <link>"                         },
335         { "modify-iptun",       do_modify_iptun,
336           "    modify-iptun     [-t] -a {local|remote}=<addr>,... <link>" },
337         { "show-iptun",         do_show_iptun,
338           "    show-iptun       [-P] [[-p] -o <field>,...] [<link>]"  },
150           "    show-iptun       [-pP] [-o <field>,..] [<link>]\n"    },
339         { "up-iptun",           do_up_iptun,            NULL        },
340         { "down-iptun",         do_down_iptun,          NULL        },
341         { "delete-phys",        do_delete_phys,
342           "    delete-phys      <link>"                              },
343         { "show-phys",          do_show_phys,
344           "    show-phys        [-P] [[-p] -o <field>,...] [-H] [<link>]" },
156           "    show-phys        [-pP] [-o <field>,..] [-H] [<link>]\n"},
345         { "init-phys",          do_init_phys,           NULL        },
346         { "show-linkmap",       do_show_linkmap,        NULL        },
347         { "create-vnic",        do_create_vnic,
348           "    create-vnic      [-t] -l <link> [-m <value> | auto |\n"
349           "\t\t     {factory [-n <slot-id>]} | {random [-r <prefix>]} |\n"
350           "\t\t     {vrrp -V <vrid> -A {inet | inet6}} [-v <vid> [-f]]\n"
351           "\t\t     [-p <prop>=<value>[,...]] [-R root-dir] <vnic-link>" },
163           "\t\t     [-p <prop>=<value>[,...]] <vnic-link>"    },
352         { "delete-vnic",        do_delete_vnic,
353           "    delete-vnic      [-t] <vnic-link>"                    },
354         { "show-vnic",          do_show_vnic,
355           "    show-vnic        [-P] [[-p] -o <field>,...] [-l <link>] "
356           "[-s [-i <interval>]] [<link>]"                             },
167           "    show-vnic        [-pP] [-l <link>] [-s [-i <interval>]] "
168           "[<link>]\n"                                               },
357         { "up-vnic",            do_up_vnic,             NULL        },
358         { "create-part",        do_create_part,
359           "    create-part      [-t] [-f] -l <link> [-P <pkey>]\n"
360           "\t\t     [-R <root-dir>] <part-link>"                    },
361         { "delete-part",        do_delete_part,
362           "    delete-part      [-t] [-R <root-dir>] <part-link>"},
363         { "show-part",          do_show_part,
364           "    show-part        [-P] [[-p] -o <field>,...] [-l <linkover>]\n"
176           "    show-part        [-pP] [-o <field>,...][-l <linkover>]\n"
365           "\t\t     [<part-link>]"          },
366         { "show-ib",            do_show_ib,
367           "    show-ib          [[-p] -o <field>,...] [<link>]"      },
179           "    show-ib          [-p] [-o <field>,...] [<link>]\n"    },
368         { "up-part",            do_up_part,             NULL        },
369         { "create-etherstub",   do_create_etherstub,
370           "    create-etherstub [-t] <link>"                        },
371         { "delete-etherstub",   do_delete_etherstub,
372           "    delete-etherstub [-t] <link>"                        },
```

```
 373         { "show-etherstub",     do_show_etherstub,
 374             "    show-etherstub   [-t] [<link>]\n"                    },
 375         { "create-simnet",      do_create_simnet,       NULL         },
 376         { "modify-simnet",      do_modify_simnet,       NULL         },
 377         { "delete-simnet",      do_delete_simnet,       NULL         },
 378         { "show-simnet",        do_show_simnet,         NULL         },
 379         { "up-simnet",          do_up_simnet,           NULL         },
 380         { "create-bridge",      do_create_bridge,
 381             "    create-bridge    [-R <root-dir>] [-P <protect>] "
 382             "[-p <priority>]\n"
 383             "\t\t     [-m <max-age>] [-h <hello-time>] [-d <forward-delay>]\n"
 384             "\t\t     [-f <force-protocol>] [-l <link>]... <bridge>"    },
 385         { "modify-bridge",      do_modify_bridge,
 386             "    modify-bridge    [-R <root-dir>] [-P <protect>] "
 387             "[-p <priority>]\n"
 388             "\t\t     [-m <max-age>] [-h <hello-time>] [-d <forward-delay>]\n"
 389             "\t\t     [-f <force-protocol>] <bridge>"                   },
 390         { "delete-bridge",      do_delete_bridge,
 391             "    delete-bridge    [-R <root-dir>] <bridge>"            },
 392         { "add-bridge",         do_add_bridge,
 393             "    add-bridge       [-R <root-dir>] -l <link> [-l <link>]... "
 394             "<bridge>"                                                 },
 395         { "remove-bridge",      do_remove_bridge,
 396             "    remove-bridge    [-R <root-dir>] -l <link> [-l <link>]... "
 397             "<bridge>"                                                 },
 398         { "show-bridge",        do_show_bridge,
 399             "    show-bridge      [[-p] -o <field>,...] [-s [-i <interval>]] "
 211             "    show-bridge      [-p] [-o <field>,...] [-s [-i <interval>]] "
 400             "[<bridge>]\n"
 401             "    show-bridge      -l [[-p] -o <field>,...] [-s [-i <interval>]]"
 213             "    show-bridge      -l [-p] [-o <field>,...] [-s [-i <interval>]]"
 402             " <bridge>\n"
 403             "    show-bridge      -f [[-p] -o <field>,...] [-s [-i <interval>]]"
 215             "    show-bridge      -f [-p] [-o <field>,...] [-s [-i <interval>]]"
 404             " <bridge>\n"
 405             "    show-bridge      -t [[-p] -o <field>,...] [-s [-i <interval>]]"
 406             " <bridge>"                                                },
 217             "    show-bridge      -t [-p] [-o <field>,...] [-s [-i <interval>]]"
 218             " <bridge>\n"                                              },
 407         { "show-usage",         do_show_usage,
 408             "    show-usage       [-a] [-d | -F <format>] "
 409             "[-s <DD/MM/YYYY,HH:MM:SS>]\n"
 410             "\t\t     [-e <DD/MM/YYYY,HH:MM:SS>] -f <logfile> [<link>]" },
 411         { "help",               do_help,
 412             "    help             [<subcommand>]"                      }
 222             "\t\t     [-e <DD/MM/YYYY,HH:MM:SS>] -f <logfile> [<link>]" }
 413 };
_____unchanged_portion_omitted_

1127 static const ofmt_field_t usage_fields[] = {
1128 { "LINK",       13,
1129         offsetof(usage_fields_buf_t, usage_link), print_default_cb},
1130 { "DURATION",   11,
1131         offsetof(usage_fields_buf_t, usage_duration), print_default_cb},
1132 { "IPACKETS",   10,
1133         offsetof(usage_fields_buf_t, usage_ipackets), print_default_cb},
1134 { "RBYTES",     11,
1135         offsetof(usage_fields_buf_t, usage_rbytes), print_default_cb},
1136 { "OPACKETS",   10,
1137         offsetof(usage_fields_buf_t, usage_opackets), print_default_cb},
1138 { "OBYTES",     11,
1139         offsetof(usage_fields_buf_t, usage_obytes), print_default_cb},
1140 { "BANDWIDTH",  15,
1141         offsetof(usage_fields_buf_t, usage_bandwidth), print_default_cb},
1142 { NULL,         0, 0, NULL}}
1143 ;
```



```
1145 /*
1146  * structures for 'dladm show-usage link'
1147  */

1149 typedef struct  usage_l_fields_buf_s {
1150         char    usage_l_link[12];
1151         char    usage_l_stime[13];
1152         char    usage_l_etime[13];
1153         char    usage_l_rbytes[8];
1154         char    usage_l_obytes[8];
1155         char    usage_l_bandwidth[14];
1156 } usage_l_fields_buf_t;
_____unchanged_portion_omitted_

1420 static ofmt_field_t bridge_trill_fields[] = {
1421 /* name,        field width,    offset, callback        */
1422 { "NICK",       5,
1423     offsetof(bridge_trill_fields_buf_t, bridget_nick), print_default_cb },
1424 { "FLAGS",      6,
1425     offsetof(bridge_trill_fields_buf_t, bridget_flags), print_default_cb },
1426 { "LINK",       12,
1427     offsetof(bridge_trill_fields_buf_t, bridget_link), print_default_cb },
1428 { "NEXTHOP",    17,
1429     offsetof(bridge_trill_fields_buf_t, bridget_nexthop), print_default_cb },
1430 { NULL,         0, 0, NULL}};

1432 static char *progname;
1433 static sig_atomic_t signalled;

1435 /*
1436  * Handle to libdladm.  Opened in main() before the sub-command
1437  * specific function is called.
1438  */
1439 static dladm_handle_t handle = NULL;

1441 #define DLADM_ETHERSTUB_NAME    "etherstub"
1442 #define DLADM_IS_ETHERSTUB(id)  (id == DATALINK_INVALID_LINKID)

1444 static void
1445 usage(void)
1446 {
1447         (void) fprintf(stderr, gettext("For more information, run: %s help\n"),
1448             progname);
1258         int     i;
1259         cmd_t   *cmdp;
1260         (void) fprintf(stderr, gettext("usage:  dladm <subcommand> <args> ..."
1261             "\n"));
1262         for (i = 0; i < sizeof (cmds) / sizeof (cmds[0]); i++) {
1263                 cmdp = &cmds[i];
1264                 if (cmdp->c_usage != NULL)
1265                         (void) fprintf(stderr, "%s\n", gettext(cmdp->c_usage));
1266         }

1450         /* close dladm handle if it was opened */
1451         if (handle != NULL)
1452                 dladm_close(handle);

1454         exit(EXIT_FAILURE);
1455 }

1457 int
1458 main(int argc, char *argv[])
1459 {
1460         int     i;
```

```
1461         cmd_t   *cmdp;
1462         dladm_status_t status;

1464         (void) setlocale(LC_ALL, "");
1465 #if !defined(TEXT_DOMAIN)
1466 #define TEXT_DOMAIN "SYS_TEST"
1467 #endif
1468         (void) textdomain(TEXT_DOMAIN);

1470         progname = argv[0];

1472         if (argc < 2) {
1473                 argv[1] = DLADM_DEFAULT_CMD;
1474                 argc++;
1475         }
1290         if (argc < 2)
1291                 usage();

1477         for (i = 0; i < sizeof (cmds) / sizeof (cmds[0]); i++) {
1478                 cmdp = &cmds[i];
1479                 if (strcmp(argv[1], cmdp->c_name) == 0) {
1480                         /* Open the libdladm handle */
1481                         if ((status = dladm_open(&handle)) != DLADM_STATUS_OK) {
1482                                 die_dlerr(status,
1483                                     "could not open /dev/dld");
1484                         }

1486                         cmdp->c_fn(argc - 1, &argv[1], cmdp->c_usage);

1488                         dladm_close(handle);
1489                         return (EXIT_SUCCESS);
1490                 }
1491         }

1493         (void) fprintf(stderr, gettext("%s: unknown subcommand '%s'\n"),
1494             progname, argv[1]);
1495         usage();
1496         return (EXIT_FAILURE);
1497 }

1499 static int
1500 help_compare(const void *cmd1, const void *cmd2)
1501 {
1502         cmd_t   *cmd1p = (cmd_t *)cmd1;
1503         cmd_t   *cmd2p = (cmd_t *)cmd2;

1505         return (strcmp(cmd1p->c_name, cmd2p->c_name));
1506 }

1508 static void
1509 do_help(int argc, char *argv[], const char *use)
1510 {
1511         size_t          nelems;
1512         int             i, j, ncols = 3;
1513         boolean_t       found = B_FALSE;

1515         _NOTE(ARGUNUSED(use));

1517         nelems = sizeof (cmds) / sizeof (cmd_t);

1519         if (argc < 2) {
1520                 qsort(cmds, nelems, sizeof (cmd_t), help_compare);

1522                 (void) fprintf(stderr, gettext(
1523                     "usage: dladm help <subcommand>\n"
1524                     "Subcommands are:\n"));
```



```
1526                 for (i = 0, j = 0; i < nelems; i++) {
1527                         if (cmds[i].c_usage == NULL)
1528                                 continue;

1530                         (void) fprintf(stderr, "%-20s", cmds[i].c_name);

1532                         if (++j % ncols == 0)
1533                                 (void) putc('\n', stderr);
1534                 }

1536                 if (j % ncols != 0)
1537                         (void) putc('\n', stderr);
1538         } else {
1539                 for (i = 0; i < nelems; i++) {
1540                         if (strcmp(argv[1], cmds[i].c_name) == 0) {
1541                                 (void) fprintf(stderr, "usage:\n%s\n",
1542                                     gettext(cmds[i].c_usage));
1543                                 found = B_TRUE;
1544                                 break;
1545                         }
1546                 }

1548                 if (!found) {
1549                         (void) fprintf(stderr, gettext(
1550                             "%s: unknown subcommand '%s'\n"),
1551                             progname, argv[1]);
1552                         usage();
1553                 }
1554         }
1555 }

1557 #endif /* ! codereview */
1558 /*ARGSUSED*/
1559 static int
1560 show_usage_date(dladm_usage_t *usage, void *arg)
1561 {
1562         show_usage_state_t      *state = (show_usage_state_t *)arg;
1563         time_t                  stime;
1564         char                    timebuf[20];
1565         dladm_status_t          status;
1566         uint32_t                flags;

1568         /*
1569          * Only show usage information for existing links unless '-a'
1570          * is specified.
1571          */
1572         if (!state->us_showall) {
1573                 if ((status = dladm_name2info(handle, usage->du_name,
1574                     NULL, &flags, NULL, NULL)) != DLADM_STATUS_OK) {
1575                         return (status);
1576                 }
1577                 if ((flags & DLADM_OPT_ACTIVE) == 0)
1578                         return (DLADM_STATUS_LINKINVAL);
1579         }

1581         stime = usage->du_stime;
1582         (void) strftime(timebuf, sizeof (timebuf), "%m/%d/%Y",
1583             localtime(&stime));
1584         (void) printf("%s\n", timebuf);

1586         return (DLADM_STATUS_OK);
1587 }

1589 static int
1590 show_usage_time(dladm_usage_t *usage, void *arg)
```

```
{
        show_usage_state_t      *state = (show_usage_state_t *)arg;
        char                    buf[DLADM_STRSIZE];
        usage_l_fields_buf_t    ubuf;
        time_t                  time;
        double                  bw;
        dladm_status_t          status;
        uint32_t                flags;

        /*
         * Only show usage information for existing links unless '-a'
         * is specified.
         */
        if (!state->us_showall) {
                if ((status = dladm_name2info(handle, usage->du_name,
                    NULL, &flags, NULL, NULL)) != DLADM_STATUS_OK) {
                        return (status);
                }
                if ((flags & DLADM_OPT_ACTIVE) == 0)
                        return (DLADM_STATUS_LINKINVAL);
        }

        if (state->us_plot) {
                if (!state->us_printheader) {
                        if (state->us_first) {
                                (void) printf("# Time");
                                state->us_first = B_FALSE;
                        }
                        (void) printf(" %s", usage->du_name);
                        if (usage->du_last) {
                                (void) printf("\n");
                                state->us_first = B_TRUE;
                                state->us_printheader = B_TRUE;
                        }
                } else {
                        if (state->us_first) {
                                time = usage->du_etime;
                                (void) strftime(buf, sizeof (buf), "%T",
                                    localtime(&time));
                                state->us_first = B_FALSE;
                                (void) printf("%s", buf);
                        }
                        bw = (double)usage->du_bandwidth/1000;
                        (void) printf(" %.2f", bw);
                        if (usage->du_last) {
                                (void) printf("\n");
                                state->us_first = B_TRUE;
                        }
                }
                return (DLADM_STATUS_OK);
        }

        bzero(&ubuf, sizeof (ubuf));

        (void) snprintf(ubuf.usage_l_link, sizeof (ubuf.usage_l_link), "%s",
            usage->du_name);
        time = usage->du_stime;
        (void) strftime(buf, sizeof (buf), "%T", localtime(&time));
        (void) snprintf(ubuf.usage_l_stime, sizeof (ubuf.usage_l_stime), "%s",
            buf);
        time = usage->du_etime;
        (void) strftime(buf, sizeof (buf), "%T", localtime(&time));
        (void) snprintf(ubuf.usage_l_etime, sizeof (ubuf.usage_l_etime), "%s",
            buf);
        (void) snprintf(ubuf.usage_l_rbytes, sizeof (ubuf.usage_l_rbytes),
            "%llu", usage->du_rbytes);
```



```
        (void) snprintf(ubuf.usage_l_obytes, sizeof (ubuf.usage_l_obytes),
            "%llu", usage->du_obytes);
        (void) snprintf(ubuf.usage_l_bandwidth, sizeof (ubuf.usage_l_bandwidth),
            "%s Mbps", dladm_bw2str(usage->du_bandwidth, buf));

        ofmt_print(state->us_ofmt, &ubuf);
        return (DLADM_STATUS_OK);
}

static int
show_usage_res(dladm_usage_t *usage, void *arg)
{
        show_usage_state_t      *state = (show_usage_state_t *)arg;
        char                    buf[DLADM_STRSIZE];
        usage_fields_buf_t      ubuf;
        dladm_status_t          status;
        uint32_t                flags;

        /*
         * Only show usage information for existing links unless '-a'
         * is specified.
         */
        if (!state->us_showall) {
                if ((status = dladm_name2info(handle, usage->du_name,
                    NULL, &flags, NULL, NULL)) != DLADM_STATUS_OK) {
                        return (status);
                }
                if ((flags & DLADM_OPT_ACTIVE) == 0)
                        return (DLADM_STATUS_LINKINVAL);
        }

        bzero(&ubuf, sizeof (ubuf));

        (void) snprintf(ubuf.usage_link, sizeof (ubuf.usage_link), "%s",
            usage->du_name);
        (void) snprintf(ubuf.usage_duration, sizeof (ubuf.usage_duration),
            "%llu", usage->du_duration);
        (void) snprintf(ubuf.usage_ipackets, sizeof (ubuf.usage_ipackets),
            "%llu", usage->du_ipackets);
        (void) snprintf(ubuf.usage_rbytes, sizeof (ubuf.usage_rbytes),
            "%llu", usage->du_rbytes);
        (void) snprintf(ubuf.usage_opackets, sizeof (ubuf.usage_opackets),
            "%llu", usage->du_opackets);
        (void) snprintf(ubuf.usage_obytes, sizeof (ubuf.usage_obytes),
            "%llu", usage->du_obytes);
        (void) snprintf(ubuf.usage_bandwidth, sizeof (ubuf.usage_bandwidth),
            "%s Mbps", dladm_bw2str(usage->du_bandwidth, buf));

        ofmt_print(state->us_ofmt, &ubuf);

        return (DLADM_STATUS_OK);
}

static boolean_t
valid_formatspec(char *formatspec_str)
{
        if (strcmp(formatspec_str, "gnuplot") == 0)
                return (B_TRUE);
        return (B_FALSE);

}

/*ARGSUSED*/
static void
do_show_usage(int argc, char *argv[], const char *use)
{
```

```
1723        char                    *file = NULL;
1724        int                     opt;
1725        dladm_status_t          status;
1726        boolean_t               d_arg = B_FALSE;
1727        char                    *stime = NULL;
1728        char                    *etime = NULL;
1729        char                    *resource = NULL;
1730        show_usage_state_t      state;
1731        boolean_t               o_arg = B_FALSE;
1732        boolean_t               F_arg = B_FALSE;
1733        char                    *fields_str = NULL;
1734        char                    *formatspec_str = NULL;
1735        char                    *all_l_fields =
1736            "link,start,end,rbytes,obytes,bandwidth";
1737        ofmt_handle_t           ofmt;
1738        ofmt_status_t           oferr;
1739        uint_t                  ofmtflags = 0;

1741        bzero(&state, sizeof (show_usage_state_t));
1742        state.us_parsable = B_FALSE;
1743        state.us_printheader = B_FALSE;
1744        state.us_plot = B_FALSE;
1745        state.us_first = B_TRUE;

1747        while ((opt = getopt_long(argc, argv, "das:e:o:f:F:",
1748            usage_opts, NULL)) != -1) {
1749                switch (opt) {
1750                case 'd':
1751                        d_arg = B_TRUE;
1752                        break;
1753                case 'a':
1754                        state.us_showall = B_TRUE;
1755                        break;
1756                case 'f':
1757                        file = optarg;
1758                        break;
1759                case 's':
1760                        stime = optarg;
1761                        break;
1762                case 'e':
1763                        etime = optarg;
1764                        break;
1765                case 'o':
1766                        o_arg = B_TRUE;
1767                        fields_str = optarg;
1768                        break;
1769                case 'F':
1770                        state.us_plot = F_arg = B_TRUE;
1771                        formatspec_str = optarg;
1772                        break;
1773                default:
1774                        die_opterr(optopt, opt, use);
1775                        break;
1776                }
1777        }

1779        if (file == NULL)
1780                die("show-usage requires a file");

1782        if (optind == (argc-1)) {
1783                uint32_t        flags;

1785                resource = argv[optind];
1786                if (!state.us_showall &&
1787                    (((status = dladm_name2info(handle, resource, NULL, &flags,
1788                        NULL, NULL)) != DLADM_STATUS_OK) ||
```



```
1789                        ((flags & DLADM_OPT_ACTIVE) == 0))) {
1790                            die("invalid link: '%s'", resource);
1791                    }
1792            }

1794            if (F_arg && d_arg)
1795                    die("incompatible -d and -F options");

1797            if (F_arg && valid_formatspec(formatspec_str) == B_FALSE)
1798                    die("Format specifier %s not supported", formatspec_str);

1800            if (state.us_parsable)
1801                    ofmtflags |= OFMT_PARSABLE;

1803            if (resource == NULL && stime == NULL && etime == NULL) {
1804                    oferr = ofmt_open(fields_str, usage_fields, ofmtflags, 0,
1805                        &ofmt);
1806            } else {
1807                    if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0))
1808                            fields_str = all_l_fields;
1809                    oferr = ofmt_open(fields_str, usage_l_fields, ofmtflags, 0,
1810                        &ofmt);

1812            }
1813            dladm_ofmt_check(oferr, state.us_parsable, ofmt);
1814            state.us_ofmt = ofmt;

1816            if (d_arg) {
1817                    /* Print log dates */
1818                    status = dladm_usage_dates(show_usage_date,
1819                        DLADM_LOGTYPE_LINK, file, resource, &state);
1820            } else if (resource == NULL && stime == NULL && etime == NULL &&
1821                !F_arg) {
1822                    /* Print summary */
1823                    status = dladm_usage_summary(show_usage_res,
1824                        DLADM_LOGTYPE_LINK, file, &state);
1825            } else if (resource != NULL) {
1826                    /* Print log entries for named resource */
1827                    status = dladm_walk_usage_res(show_usage_time,
1828                        DLADM_LOGTYPE_LINK, file, resource, stime, etime, &state);
1829            } else {
1830                    /* Print time and information for each link */
1831                    status = dladm_walk_usage_time(show_usage_time,
1832                        DLADM_LOGTYPE_LINK, file, stime, etime, &state);
1833            }

1835            if (status != DLADM_STATUS_OK)
1836                    die_dlerr(status, "show-usage");
1837            ofmt_close(ofmt);
1838    }

1840    static void
1841    do_create_aggr(int argc, char *argv[], const char *use)
1842    {
1843            int                     option;
1844            int                     key = 0;
1845            uint32_t                policy = AGGR_POLICY_L4;
1846            aggr_lacp_mode_t        lacp_mode = AGGR_LACP_OFF;
1847            aggr_lacp_timer_t       lacp_timer = AGGR_LACP_TIMER_SHORT;
1848            dladm_aggr_port_attr_db_t       port[MAXPORT];
1849            uint_t                  n, ndev, nlink;
1850            uint8_t                 mac_addr[ETHERADDRL];
1851            boolean_t               mac_addr_fixed = B_FALSE;
1852            boolean_t               P_arg = B_FALSE;
1853            boolean_t               l_arg = B_FALSE;
1854            boolean_t               u_arg = B_FALSE;
```

```
1855        boolean_t               T_arg = B_FALSE;
1856        uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
1857        char                    *altroot = NULL;
1858        char                    name[MAXLINKNAMELEN];
1859        char                    *devs[MAXPORT];
1860        char                    *links[MAXPORT];
1861        dladm_status_t          status;
1862        dladm_status_t          pstatus;
1863        char                    propstr[DLADM_STRSIZE];
1864        dladm_arg_list_t        *proplist = NULL;
1865        int                     i;
1866        datalink_id_t           linkid;

1868        ndev = nlink = opterr = 0;
1869        bzero(propstr, DLADM_STRSIZE);

1871        while ((option = getopt_long(argc, argv, ":d:l:L:P:R:tfu:T:p:",
1872            lopts, NULL)) != -1) {
1873                switch (option) {
1874                case 'd':
1875                        if (ndev + nlink >= MAXPORT)
1876                                die("too many ports specified");

1878                        devs[ndev++] = optarg;
1879                        break;
1880                case 'P':
1881                        if (P_arg)
1882                                die_optdup(option);

1884                        P_arg = B_TRUE;
1885                        if (!dladm_aggr_str2policy(optarg, &policy))
1886                                die("invalid policy '%s'", optarg);
1887                        break;
1888                case 'u':
1889                        if (u_arg)
1890                                die_optdup(option);

1892                        u_arg = B_TRUE;
1893                        if (!dladm_aggr_str2macaddr(optarg, &mac_addr_fixed,
1894                            mac_addr))
1895                                die("invalid MAC address '%s'", optarg);
1896                        break;
1897                case 'l':
1898                        if (isdigit(optarg[strlen(optarg) - 1])) {

1900                                /*
1901                                 * Ended with digit, possibly a link name.
1902                                 */
1903                                if (ndev + nlink >= MAXPORT)
1904                                        die("too many ports specified");

1906                                links[nlink++] = optarg;
1907                                break;
1908                        }
1909                        /* FALLTHROUGH */
1910                case 'L':
1911                        if (l_arg)
1912                                die_optdup(option);

1914                        l_arg = B_TRUE;
1915                        if (!dladm_aggr_str2lacpmode(optarg, &lacp_mode))
1916                                die("invalid LACP mode '%s'", optarg);
1917                        break;
1918                case 'T':
1919                        if (T_arg)
1920                                die_optdup(option);
```



```
1922                        T_arg = B_TRUE;
1923                        if (!dladm_aggr_str2lacptimer(optarg, &lacp_timer))
1924                                die("invalid LACP timer value '%s'", optarg);
1925                        break;
1926                case 't':
1927                        flags &= ~DLADM_OPT_PERSIST;
1928                        break;
1929                case 'f':
1930                        flags |= DLADM_OPT_FORCE;
1931                        break;
1932                case 'R':
1933                        altroot = optarg;
1934                        break;
1935                case 'p':
1936                        (void) strlcat(propstr, optarg, DLADM_STRSIZE);
1937                        if (strlcat(propstr, ",", DLADM_STRSIZE) >=
1938                            DLADM_STRSIZE)
1939                                die("property list too long '%s'", propstr);
1940                        break;

1942                default:
1943                        die_opterr(optopt, option, use);
1944                        break;
1945                }
1946        }

1948        if (ndev + nlink == 0)
1949                usage();

1951        /* get key value or the aggregation name (required last argument) */
1952        if (optind != (argc-1))
1953                usage();

1955        if (!str2int(argv[optind], &key)) {
1956                if (strlcpy(name, argv[optind], MAXLINKNAMELEN) >=
1957                    MAXLINKNAMELEN) {
1958                        die("link name too long '%s'", argv[optind]);
1959                }

1961                if (!dladm_valid_linkname(name))
1962                        die("invalid link name '%s'", argv[optind]);
1963        } else {
1964                (void) snprintf(name, MAXLINKNAMELEN, "aggr%d", key);
1965        }

1967        if (altroot != NULL)
1968                altroot_cmd(altroot, argc, argv);

1970        for (n = 0; n < ndev; n++) {
1971                if ((status = dladm_dev2linkid(handle, devs[n],
1972                    &port[n].lp_linkid)) != DLADM_STATUS_OK) {
1973                        die_dlerr(status, "invalid dev name '%s'", devs[n]);
1974                }
1975        }

1977        for (n = 0; n < nlink; n++) {
1978                if ((status = dladm_name2info(handle, links[n],
1979                    &port[ndev + n].lp_linkid, NULL, NULL, NULL)) !=
1980                    DLADM_STATUS_OK) {
1981                        die_dlerr(status, "invalid link name '%s'", links[n]);
1982                }
1983        }

1985        status = dladm_aggr_create(handle, name, key, ndev + nlink, port,
1986            policy, mac_addr_fixed, (const uchar_t *)mac_addr, lacp_mode,
```

```
1987             lacp_timer, flags);
1988         if (status != DLADM_STATUS_OK)
1989                 goto done;

1991         if (dladm_parse_link_props(propstr, &proplist, B_FALSE)
1992             != DLADM_STATUS_OK)
1993                 die("invalid aggregation property");

1995         if (proplist == NULL)
1996                 return;

1998         status = dladm_name2info(handle, name, &linkid, NULL, NULL, NULL);
1999         if (status != DLADM_STATUS_OK)
2000                 goto done;

2002         for (i = 0; i < proplist->al_count; i++) {
2003                 dladm_arg_info_t        *aip = &proplist->al_info[i];

2005                 pstatus = dladm_set_linkprop(handle, linkid, aip->ai_name,
2006                     aip->ai_val, aip->ai_count, flags);

2008                 if (pstatus != DLADM_STATUS_OK) {
2009                         die_dlerr(pstatus,
2010                             "aggr creation succeeded but "
2011                             "could not set property '%s'", aip->ai_name);
2012                 }
2013         }
2014 done:
2015         dladm_free_props(proplist);
2016         if (status != DLADM_STATUS_OK) {
2017                 if (status == DLADM_STATUS_NONOTIF) {
2018                         die("not all links have link up/down detection; must "
2019                             "use -f (see dladm(1M))");
2020                 } else {
2021                         die_dlerr(status, "create operation failed");
2022                 }
2023         }
2024 }

2026 /*
2027  * arg is either the key or the aggr name. Validate it and convert it to
2028  * the linkid if altroot is NULL.
2029  */
2030 static dladm_status_t
2031 i_dladm_aggr_get_linkid(const char *altroot, const char *arg,
2032     datalink_id_t *linkidp, uint32_t flags)
2033 {
2034         int             key = 0;
2035         char            *aggr = NULL;
2036         dladm_status_t  status;

2038         if (!str2int(arg, &key))
2039                 aggr = (char *)arg;

2041         if (aggr == NULL && key == 0)
2042                 return (DLADM_STATUS_LINKINVAL);

2044         if (altroot != NULL)
2045                 return (DLADM_STATUS_OK);

2047         if (aggr != NULL) {
2048                 status = dladm_name2info(handle, aggr, linkidp, NULL, NULL,
2049                     NULL);
2050         } else {
2051                 status = dladm_key2linkid(handle, key, linkidp, flags);
2052         }
```



```
2054         return (status);
2055 }

2057 static void
2058 do_delete_aggr(int argc, char *argv[], const char *use)
2059 {
2060         int                     option;
2061         char                    *altroot = NULL;
2062         uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
2063         dladm_status_t          status;
2064         datalink_id_t           linkid;

2066         opterr = 0;
2067         while ((option = getopt_long(argc, argv, ":R:t", lopts, NULL)) != -1) {
2068                 switch (option) {
2069                 case 't':
2070                         flags &= ~DLADM_OPT_PERSIST;
2071                         break;
2072                 case 'R':
2073                         altroot = optarg;
2074                         break;
2075                 default:
2076                         die_opterr(optopt, option, use);
2077                         break;
2078                 }
2079         }

2081         /* get key value or the aggregation name (required last argument) */
2082         if (optind != (argc-1))
2083                 usage();

2085         status = i_dladm_aggr_get_linkid(altroot, argv[optind], &linkid, flags);
2086         if (status != DLADM_STATUS_OK)
2087                 goto done;

2089         if (altroot != NULL)
2090                 altroot_cmd(altroot, argc, argv);

2092         status = dladm_aggr_delete(handle, linkid, flags);
2093 done:
2094         if (status != DLADM_STATUS_OK)
2095                 die_dlerr(status, "delete operation failed");
2096 }

2098 static void
2099 do_add_aggr(int argc, char *argv[], const char *use)
2100 {
2101         int                     option;
2102         uint_t                  n, ndev, nlink;
2103         char                    *altroot = NULL;
2104         uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
2105         datalink_id_t           linkid;
2106         dladm_status_t          status;
2107         dladm_aggr_port_attr_db_t       port[MAXPORT];
2108         char                    *devs[MAXPORT];
2109         char                    *links[MAXPORT];

2111         ndev = nlink = opterr = 0;
2112         while ((option = getopt_long(argc, argv, ":d:l:R:tf", lopts,
2113             NULL)) != -1) {
2114                 switch (option) {
2115                 case 'd':
2116                         if (ndev + nlink >= MAXPORT)
2117                                 die("too many ports specified");
```

```
2119                        devs[ndev++] = optarg;
2120                        break;
2121                case 'l':
2122                        if (ndev + nlink >= MAXPORT)
2123                                die("too many ports specified");
2125                        links[nlink++] = optarg;
2126                        break;
2127                case 't':
2128                        flags &= ~DLADM_OPT_PERSIST;
2129                        break;
2130                case 'f':
2131                        flags |= DLADM_OPT_FORCE;
2132                        break;
2133                case 'R':
2134                        altroot = optarg;
2135                        break;
2136                default:
2137                        die_opterr(optopt, option, use);
2138                        break;
2139                }
2140        }
2142        if (ndev + nlink == 0)
2143                usage();
2145        /* get key value or the aggregation name (required last argument) */
2146        if (optind != (argc-1))
2147                usage();
2149        if ((status = i_dladm_aggr_get_linkid(altroot, argv[optind], &linkid,
2150            flags & (DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST))) !=
2151            DLADM_STATUS_OK) {
2152                goto done;
2153        }
2155        if (altroot != NULL)
2156                altroot_cmd(altroot, argc, argv);
2158        for (n = 0; n < ndev; n++) {
2159                if ((status = dladm_dev2linkid(handle, devs[n],
2160                    &(port[n].lp_linkid))) != DLADM_STATUS_OK) {
2161                        die_dlerr(status, "invalid <dev> '%s'", devs[n]);
2162                }
2163        }
2165        for (n = 0; n < nlink; n++) {
2166                if ((status = dladm_name2info(handle, links[n],
2167                    &port[n + ndev].lp_linkid, NULL, NULL, NULL)) !=
2168                    DLADM_STATUS_OK) {
2169                        die_dlerr(status, "invalid <link> '%s'", links[n]);
2170                }
2171        }
2173        status = dladm_aggr_add(handle, linkid, ndev + nlink, port, flags);
2174 done:
2175        if (status != DLADM_STATUS_OK) {
2176                /*
2177                 * checking DLADM_STATUS_NOTSUP is a temporary workaround
2178                 * and should be removed once 6399681 is fixed.
2179                 */
2180                if (status == DLADM_STATUS_NOTSUP) {
2181                        die("add operation failed: link capabilities don't "
2182                            "match");
2183                } else if (status == DLADM_STATUS_NONOTIF) {
2184                        die("not all links have link up/down detection; must "
```



```
2185                            "use -f (see dladm(1M))");
2186                } else {
2187                        die_dlerr(status, "add operation failed");
2188                }
2189        }
2190 }
2192 static void
2193 do_remove_aggr(int argc, char *argv[], const char *use)
2194 {
2195        int                             option;
2196        dladm_aggr_port_attr_db_t       port[MAXPORT];
2197        uint_t                          n, ndev, nlink;
2198        char                            *devs[MAXPORT];
2199        char                            *links[MAXPORT];
2200        char                            *altroot = NULL;
2201        uint32_t                        flags;
2202        datalink_id_t                   linkid;
2203        dladm_status_t                  status;
2205        flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
2206        ndev = nlink = opterr = 0;
2207        while ((option = getopt_long(argc, argv, ":d:l:R:t",
2208            lopts, NULL)) != -1) {
2209                switch (option) {
2210                case 'd':
2211                        if (ndev + nlink >= MAXPORT)
2212                                die("too many ports specified");
2214                        devs[ndev++] = optarg;
2215                        break;
2216                case 'l':
2217                        if (ndev + nlink >= MAXPORT)
2218                                die("too many ports specified");
2220                        links[nlink++] = optarg;
2221                        break;
2222                case 't':
2223                        flags &= ~DLADM_OPT_PERSIST;
2224                        break;
2225                case 'R':
2226                        altroot = optarg;
2227                        break;
2228                default:
2229                        die_opterr(optopt, option, use);
2230                        break;
2231                }
2232        }
2234        if (ndev + nlink == 0)
2235                usage();
2237        /* get key value or the aggregation name (required last argument) */
2238        if (optind != (argc-1))
2239                usage();
2241        status = i_dladm_aggr_get_linkid(altroot, argv[optind], &linkid, flags);
2242        if (status != DLADM_STATUS_OK)
2243                goto done;
2245        if (altroot != NULL)
2246                altroot_cmd(altroot, argc, argv);
2248        for (n = 0; n < ndev; n++) {
2249                if ((status = dladm_dev2linkid(handle, devs[n],
2250                    &(port[n].lp_linkid))) != DLADM_STATUS_OK) {
```

```
2251                        die_dlerr(status, "invalid <dev> '%s'", devs[n]);
2252                }
2253        }

2255        for (n = 0; n < nlink; n++) {
2256                if ((status = dladm_name2info(handle, links[n],
2257                    &port[n + ndev].lp_linkid, NULL, NULL, NULL)) !=
2258                    DLADM_STATUS_OK) {
2259                        die_dlerr(status, "invalid <link> '%s'", links[n]);
2260                }
2261        }

2263        status = dladm_aggr_remove(handle, linkid, ndev + nlink, port, flags);
2264 done:
2265        if (status != DLADM_STATUS_OK)
2266                die_dlerr(status, "remove operation failed");
2267 }

2269 static void
2270 do_modify_aggr(int argc, char *argv[], const char *use)
2271 {
2272        int                     option;
2273        uint32_t                policy = AGGR_POLICY_L4;
2274        aggr_lacp_mode_t        lacp_mode = AGGR_LACP_OFF;
2275        aggr_lacp_timer_t       lacp_timer = AGGR_LACP_TIMER_SHORT;
2276        uint8_t                 mac_addr[ETHERADDRL];
2277        boolean_t               mac_addr_fixed = B_FALSE;
2278        uint8_t                 modify_mask = 0;
2279        char                    *altroot = NULL;
2280        uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
2281        datalink_id_t           linkid;
2282        dladm_status_t          status;

2284        opterr = 0;
2285        while ((option = getopt_long(argc, argv, ":L:l:P:R:tu:T:", lopts,
2286            NULL)) != -1) {
2287                switch (option) {
2288                case 'P':
2289                        if (modify_mask & DLADM_AGGR_MODIFY_POLICY)
2290                                die_optdup(option);

2292                        modify_mask |= DLADM_AGGR_MODIFY_POLICY;

2294                        if (!dladm_aggr_str2policy(optarg, &policy))
2295                                die("invalid policy '%s'", optarg);
2296                        break;
2297                case 'u':
2298                        if (modify_mask & DLADM_AGGR_MODIFY_MAC)
2299                                die_optdup(option);

2301                        modify_mask |= DLADM_AGGR_MODIFY_MAC;

2303                        if (!dladm_aggr_str2macaddr(optarg, &mac_addr_fixed,
2304                            mac_addr))
2305                                die("invalid MAC address '%s'", optarg);
2306                        break;
2307                case 'l':
2308                case 'L':
2309                        if (modify_mask & DLADM_AGGR_MODIFY_LACP_MODE)
2310                                die_optdup(option);

2312                        modify_mask |= DLADM_AGGR_MODIFY_LACP_MODE;

2314                        if (!dladm_aggr_str2lacpmode(optarg, &lacp_mode))
2315                                die("invalid LACP mode '%s'", optarg);
2316                        break;
```



```
2317                case 'T':
2318                        if (modify_mask & DLADM_AGGR_MODIFY_LACP_TIMER)
2319                                die_optdup(option);

2321                        modify_mask |= DLADM_AGGR_MODIFY_LACP_TIMER;

2323                        if (!dladm_aggr_str2lacptimer(optarg, &lacp_timer))
2324                                die("invalid LACP timer value '%s'", optarg);
2325                        break;
2326                case 't':
2327                        flags &= ~DLADM_OPT_PERSIST;
2328                        break;
2329                case 'R':
2330                        altroot = optarg;
2331                        break;
2332                default:
2333                        die_opterr(optopt, option, use);
2334                        break;
2335                }
2336        }

2338        if (modify_mask == 0)
2339                die("at least one of the -PulT options must be specified");

2341        /* get key value or the aggregation name (required last argument) */
2342        if (optind != (argc-1))
2343                usage();

2345        status = i_dladm_aggr_get_linkid(altroot, argv[optind], &linkid, flags);
2346        if (status != DLADM_STATUS_OK)
2347                goto done;

2349        if (altroot != NULL)
2350                altroot_cmd(altroot, argc, argv);

2352        status = dladm_aggr_modify(handle, linkid, modify_mask, policy,
2353            mac_addr_fixed, (const uchar_t *)mac_addr, lacp_mode, lacp_timer,
2354            flags);

2356 done:
2357        if (status != DLADM_STATUS_OK)
2358                die_dlerr(status, "modify operation failed");
2359 }

2361 /*ARGSUSED*/
2362 static void
2363 do_up_aggr(int argc, char *argv[], const char *use)
2364 {
2365        datalink_id_t   linkid = DATALINK_ALL_LINKID;
2366        dladm_status_t  status;

2368        /*
2369         * get the key or the name of the aggregation (optional last argument)
2370         */
2371        if (argc == 2) {
2372                if ((status = i_dladm_aggr_get_linkid(NULL, argv[1], &linkid,
2373                    DLADM_OPT_PERSIST)) != DLADM_STATUS_OK)
2374                        goto done;
2375        } else if (argc > 2) {
2376                usage();
2377        }

2379        status = dladm_aggr_up(handle, linkid);
2380 done:
2381        if (status != DLADM_STATUS_OK) {
2382                if (argc == 2) {
```

```
2383                        die_dlerr(status,
2384                            "could not bring up aggregation '%s'", argv[1]);
2385                } else {
2386                        die_dlerr(status, "could not bring aggregations up");
2387                }
2388        }
2389 }

2391 static void
2392 do_create_vlan(int argc, char *argv[], const char *use)
2393 {
2394        char                    *link = NULL;
2395        char                    drv[DLPI_LINKNAME_MAX];
2396        uint_t                  ppa;
2397        datalink_id_t           linkid;
2398        datalink_id_t           dev_linkid;
2399        int                     vid = 0;
2400        int                     option;
2401        uint32_t                flags = (DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST);
2402        char                    *altroot = NULL;
2403        char                    vlan[MAXLINKNAMELEN];
2404        char                    propstr[DLADM_STRSIZE];
2405        dladm_arg_list_t        *proplist = NULL;
2406        dladm_status_t          status;

2408        opterr = 0;
2409        bzero(propstr, DLADM_STRSIZE);

2411        while ((option = getopt_long(argc, argv, ":tfR:l:v:p:",
2412            lopts, NULL)) != -1) {
2413                switch (option) {
2414                case 'v':
2415                        if (vid != 0)
2416                                die_optdup(option);

2418                        if (!str2int(optarg, &vid) || vid < 1 || vid > 4094)
2419                                die("invalid VLAN identifier '%s'", optarg);

2421                        break;
2422                case 'l':
2423                        if (link != NULL)
2424                                die_optdup(option);

2426                        link = optarg;
2427                        break;
2428                case 't':
2429                        flags &= ~DLADM_OPT_PERSIST;
2430                        break;
2431                case 'R':
2432                        altroot = optarg;
2433                        break;
2434                case 'p':
2435                        (void) strlcat(propstr, optarg, DLADM_STRSIZE);
2436                        if (strlcat(propstr, ",", DLADM_STRSIZE) >=
2437                            DLADM_STRSIZE)
2438                                die("property list too long '%s'", propstr);
2439                        break;
2440                case 'f':
2441                        flags |= DLADM_OPT_FORCE;
2442                        break;
2443                default:
2444                        die_opterr(optopt, option, use);
2445                        break;
2446                }
2447        }
```



```
2449        /* get vlan name if there is any */
2450        if ((vid == 0) || (link == NULL) || (argc - optind > 1))
2451                usage();

2453        if (optind == (argc - 1)) {
2454                if (strlcpy(vlan, argv[optind], MAXLINKNAMELEN) >=
2455                    MAXLINKNAMELEN) {
2456                        die("vlan name too long '%s'", argv[optind]);
2457                }
2458        } else {
2459                if ((dlpi_parselink(link, drv, &ppa) != DLPI_SUCCESS) ||
2460                    (ppa >= 1000) ||
2461                    (dlpi_makelink(vlan, drv, vid * 1000 + ppa) !=
2462                    DLPI_SUCCESS)) {
2463                        die("invalid link name '%s'", link);
2464                }
2465        }

2467        if (altroot != NULL)
2468                altroot_cmd(altroot, argc, argv);

2470        if (dladm_name2info(handle, link, &dev_linkid, NULL, NULL, NULL) !=
2471            DLADM_STATUS_OK) {
2472                die("invalid link name '%s'", link);
2473        }

2475        if (dladm_parse_link_props(propstr, &proplist, B_FALSE)
2476            != DLADM_STATUS_OK)
2477                die("invalid vlan property");

2479        status = dladm_vlan_create(handle, vlan, dev_linkid, vid, proplist,
2480            flags, &linkid);
2481        switch (status) {
2482        case DLADM_STATUS_OK:
2483                break;

2485        case DLADM_STATUS_NOTSUP:
2486                die("VLAN over '%s' may require lowered MTU; must use -f (see "
2487                    "dladm(1M))", link);
2488                break;

2490        case DLADM_STATUS_LINKBUSY:
2491                die("VLAN over '%s' may not use default_tag ID "
2492                    "(see dladm(1M))", link);
2493                break;

2495        default:
2496                die_dlerr(status, "create operation failed");
2497        }
2498 }

2500 static void
2501 do_delete_vlan(int argc, char *argv[], const char *use)
2502 {
2503        int             option;
2504        uint32_t        flags = (DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST);
2505        char            *altroot = NULL;
2506        datalink_id_t   linkid;
2507        dladm_status_t  status;

2509        opterr = 0;
2510        while ((option = getopt_long(argc, argv, ":R:t", lopts, NULL)) != -1) {
2511                switch (option) {
2512                case 't':
2513                        flags &= ~DLADM_OPT_PERSIST;
2514                        break;
```

```
2515                 case 'R':
2516                         altroot = optarg;
2517                         break;
2518                 default:
2519                         die_opterr(optopt, option, use);
2520                         break;
2521                 }
2522         }

2524         /* get VLAN link name (required last argument) */
2525         if (optind != (argc - 1))
2526                 usage();

2528         if (altroot != NULL)
2529                 altroot_cmd(altroot, argc, argv);

2531         status = dladm_name2info(handle, argv[optind], &linkid, NULL, NULL,
2532             NULL);
2533         if (status != DLADM_STATUS_OK)
2534                 goto done;

2536         status = dladm_vlan_delete(handle, linkid, flags);
2537 done:
2538         if (status != DLADM_STATUS_OK)
2539                 die_dlerr(status, "delete operation failed");
2540 }

2542 /*ARGSUSED*/
2543 static void
2544 do_up_vlan(int argc, char *argv[], const char *use)
2545 {
2546         do_up_vnic_common(argc, argv, use, B_TRUE);
2547 }

2549 static void
2550 do_rename_link(int argc, char *argv[], const char *use)
2551 {
2552         int             option;
2553         char            *link1, *link2;
2554         char            *altroot = NULL;
2555         dladm_status_t  status;

2557         opterr = 0;
2558         while ((option = getopt_long(argc, argv, ":R:", lopts, NULL)) != -1) {
2559                 switch (option) {
2560                 case 'R':
2561                         altroot = optarg;
2562                         break;
2563                 default:
2564                         die_opterr(optopt, option, use);
2565                         break;
2566                 }
2567         }

2569         /* get link1 and link2 name (required the last 2 arguments) */
2570         if (optind != (argc - 2))
2571                 usage();

2573         if (altroot != NULL)
2574                 altroot_cmd(altroot, argc, argv);

2576         link1 = argv[optind++];
2577         link2 = argv[optind];
2578         if ((status = dladm_rename_link(handle, link1, link2)) !=
2579             DLADM_STATUS_OK)
2580                 die_dlerr(status, "rename operation failed");
```



```
2581 }

2583 /*ARGSUSED*/
2584 static void
2585 do_delete_phys(int argc, char *argv[], const char *use)
2586 {
2587         datalink_id_t   linkid = DATALINK_ALL_LINKID;
2588         dladm_status_t  status;

2590         /* get link name (required the last argument) */
2591         if (argc > 2)
2592                 usage();

2594         if (argc == 2) {
2595                 if ((status = dladm_name2info(handle, argv[1], &linkid, NULL,
2596                     NULL, NULL)) != DLADM_STATUS_OK)
2597                         die_dlerr(status, "cannot delete '%s'", argv[1]);
2598         }

2600         if ((status = dladm_phys_delete(handle, linkid)) != DLADM_STATUS_OK) {
2601                 if (argc == 2)
2602                         die_dlerr(status, "cannot delete '%s'", argv[1]);
2603                 else
2604                         die_dlerr(status, "delete operation failed");
2605         }
2606 }

2608 /*ARGSUSED*/
2609 static int
2610 i_dladm_walk_linkmap(dladm_handle_t dh, datalink_id_t linkid, void *arg)
2611 {
2612         char                    name[MAXLINKNAMELEN];
2613         char                    mediabuf[DLADM_STRSIZE];
2614         char                    classbuf[DLADM_STRSIZE];
2615         datalink_class_t        class;
2616         uint32_t                media;
2617         uint32_t                flags;

2619         if (dladm_datalink_id2info(dh, linkid, &flags, &class, &media, name,
2620             MAXLINKNAMELEN) == DLADM_STATUS_OK) {
2621                 (void) dladm_class2str(class, classbuf);
2622                 (void) dladm_media2str(media, mediabuf);
2623                 (void) printf("%-12s%8d  %-12s%-20s %6d\n", name,
2624                     linkid, classbuf, mediabuf, flags);
2625         }
2626         return (DLADM_WALK_CONTINUE);
2627 }

2629 /*ARGSUSED*/
2630 static void
2631 do_show_linkmap(int argc, char *argv[], const char *use)
2632 {
2633         if (argc != 1)
2634                 die("invalid arguments");

2636         (void) printf("%-12s%8s  %-12s%-20s %6s\n", "NAME", "LINKID",
2637             "CLASS", "MEDIA", "FLAGS");

2639         (void) dladm_walk_datalink_id(i_dladm_walk_linkmap, handle, NULL,
2640             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE,
2641             DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST);
2642 }

2644 /*
2645  * Delete inactive physical links.
2646  */
```

```
2647 /*ARGSUSED*/
2648 static int
2649 purge_phys(dladm_handle_t dh, datalink_id_t linkid, void *arg)
2650 {
2651         datalink_class_t        class;
2652         uint32_t                flags;

2654         if (dladm_datalink_id2info(dh, linkid, &flags, &class, NULL, NULL, 0)
2655             != DLADM_STATUS_OK) {
2656                 return (DLADM_WALK_CONTINUE);
2657         }

2659         if (class == DATALINK_CLASS_PHYS && !(flags & DLADM_OPT_ACTIVE))
2660                 (void) dladm_phys_delete(dh, linkid);

2662         return (DLADM_WALK_CONTINUE);
2663 }

2665 /*ARGSUSED*/
2666 static void
2667 do_init_phys(int argc, char *argv[], const char *use)
2668 {
2669         di_node_t       devtree;

2671         if (argc > 1)
2672                 usage();

2674         /*
2675          * Force all the devices to attach, therefore all the network physical
2676          * devices can be known to the dlmgmtd daemon.
2677          */
2678         if ((devtree = di_init("/", DINFOFORCE | DINFOSUBTREE)) != DI_NODE_NIL)
2679                 di_fini(devtree);

2681         (void) dladm_walk_datalink_id(purge_phys, handle, NULL,
2682             DATALINK_CLASS_PHYS, DATALINK_ANY_MEDIATYPE, DLADM_OPT_PERSIST);
2683 }

2685 /*
2686  * Print the active topology information.
2687  */
2688 void
2689 print_link_topology(show_state_t *state, datalink_id_t linkid,
2690     datalink_class_t class, link_fields_buf_t *lbuf)
2691 {
2692         uint32_t        flags = state->ls_flags;
2693         dladm_status_t  status;
2694         char            tmpbuf[MAXLINKNAMELEN];

2696         lbuf->link_over[0] = '\0';
2697         lbuf->link_bridge[0] = '\0';

2699         switch (class) {
2700         case DATALINK_CLASS_AGGR:
2701         case DATALINK_CLASS_PHYS:
2702         case DATALINK_CLASS_ETHERSTUB:
2703                 status = dladm_bridge_getlink(handle, linkid, lbuf->link_bridge,
2704                     sizeof (lbuf->link_bridge));
2705                 if (status != DLADM_STATUS_OK &&
2706                     status != DLADM_STATUS_NOTFOUND)
2707                         (void) strcpy(lbuf->link_bridge, "?");
2708                 break;
2709         }

2711         switch (class) {
2712         case DATALINK_CLASS_VLAN: {
```



```
2713                 dladm_vlan_attr_t       vinfo;

2715                 if (dladm_vlan_info(handle, linkid, &vinfo, flags) !=
2716                     DLADM_STATUS_OK) {
2717                         (void) strcpy(lbuf->link_over, "?");
2718                         break;
2719                 }
2720                 if (dladm_datalink_id2info(handle, vinfo.dv_linkid, NULL, NULL,
2721                     NULL, lbuf->link_over, sizeof (lbuf->link_over)) !=
2722                     DLADM_STATUS_OK)
2723                         (void) strcpy(lbuf->link_over, "?");
2724                 break;
2725         }
2726         case DATALINK_CLASS_AGGR: {
2727                 dladm_aggr_grp_attr_t   ginfo;
2728                 int                     i;

2730                 if (dladm_aggr_info(handle, linkid, &ginfo, flags) !=
2731                     DLADM_STATUS_OK || ginfo.lg_nports == 0) {
2732                         (void) strcpy(lbuf->link_over, "?");
2733                         break;
2734                 }
2735                 for (i = 0; i < ginfo.lg_nports; i++) {
2736                         if (dladm_datalink_id2info(handle,
2737                             ginfo.lg_ports[i].lp_linkid, NULL, NULL, NULL,
2738                             tmpbuf, sizeof (tmpbuf)) != DLADM_STATUS_OK) {
2739                                 (void) strcpy(lbuf->link_over, "?");
2740                                 break;
2741                         }
2742                         (void) strlcat(lbuf->link_over, tmpbuf,
2743                             sizeof (lbuf->link_over));
2744                         if (i != (ginfo.lg_nports - 1)) {
2745                                 (void) strlcat(lbuf->link_over, " ",
2746                                     sizeof (lbuf->link_over));
2747                         }
2748                 }
2749                 free(ginfo.lg_ports);
2750                 break;
2751         }
2752         case DATALINK_CLASS_VNIC: {
2753                 dladm_vnic_attr_t       vinfo;

2755                 if (dladm_vnic_info(handle, linkid, &vinfo, flags) !=
2756                     DLADM_STATUS_OK) {
2757                         (void) strcpy(lbuf->link_over, "?");
2758                         break;
2759                 }
2760                 if (dladm_datalink_id2info(handle, vinfo.va_link_id, NULL, NULL,
2761                     NULL, lbuf->link_over, sizeof (lbuf->link_over)) !=
2762                     DLADM_STATUS_OK)
2763                         (void) strcpy(lbuf->link_over, "?");
2764                 break;
2765         }

2767         case DATALINK_CLASS_PART: {
2768                 dladm_part_attr_t       pinfo;

2770                 if (dladm_part_info(handle, linkid, &pinfo, flags) !=
2771                     DLADM_STATUS_OK) {
2772                         (void) strcpy(lbuf->link_over, "?");
2773                         break;
2774                 }
2775                 if (dladm_datalink_id2info(handle, pinfo.dia_physlinkid, NULL,
2776                     NULL, NULL, lbuf->link_over, sizeof (lbuf->link_over)) !=
2777                     DLADM_STATUS_OK)
2778                         (void) strcpy(lbuf->link_over, "?");
```

```
2779                    break;
2780            }

2782            case DATALINK_CLASS_BRIDGE: {
2783                    datalink_id_t *dlp;
2784                    uint_t i, nports;

2786                    if (dladm_datalink_id2info(handle, linkid, NULL, NULL,
2787                        NULL, tmpbuf, sizeof (tmpbuf)) != DLADM_STATUS_OK) {
2788                            (void) strcpy(lbuf->link_over, "?");
2789                            break;
2790                    }
2791                    if (tmpbuf[0] != '\0')
2792                            tmpbuf[strlen(tmpbuf) - 1] = '\0';
2793                    dlp = dladm_bridge_get_portlist(tmpbuf, &nports);
2794                    if (dlp == NULL) {
2795                            (void) strcpy(lbuf->link_over, "?");
2796                            break;
2797                    }
2798                    for (i = 0; i < nports; i++) {
2799                            if (dladm_datalink_id2info(handle, dlp[i], NULL,
2800                                NULL, NULL, tmpbuf, sizeof (tmpbuf)) !=
2801                                DLADM_STATUS_OK) {
2802                                    (void) strcpy(lbuf->link_over, "?");
2803                                    break;
2804                            }
2805                            (void) strlcat(lbuf->link_over, tmpbuf,
2806                                sizeof (lbuf->link_over));
2807                            if (i != nports - 1) {
2808                                    (void) strlcat(lbuf->link_over, " ",
2809                                        sizeof (lbuf->link_over));
2810                            }
2811                    }
2812                    dladm_bridge_free_portlist(dlp);
2813                    break;
2814            }

2816            case DATALINK_CLASS_SIMNET: {
2817                    dladm_simnet_attr_t     slinfo;

2819                    if (dladm_simnet_info(handle, linkid, &slinfo, flags) !=
2820                        DLADM_STATUS_OK) {
2821                            (void) strcpy(lbuf->link_over, "?");
2822                            break;
2823                    }
2824                    if (slinfo.sna_peer_link_id != DATALINK_INVALID_LINKID) {
2825                            if (dladm_datalink_id2info(handle,
2826                                slinfo.sna_peer_link_id, NULL, NULL, NULL,
2827                                lbuf->link_over, sizeof (lbuf->link_over)) !=
2828                                DLADM_STATUS_OK)
2829                                    (void) strcpy(lbuf->link_over, "?");
2830                    }
2831                    break;
2832            }
2833            }
2834 }

2836 static dladm_status_t
2837 print_link(show_state_t *state, datalink_id_t linkid, link_fields_buf_t *lbuf)
2838 {
2839            char                    link[MAXLINKNAMELEN];
2840            datalink_class_t        class;
2841            uint_t                  mtu;
2842            uint32_t                flags;
2843            dladm_status_t          status;
```



```
2845            if ((status = dladm_datalink_id2info(handle, linkid, &flags, &class,
2846                NULL, link, sizeof (link))) != DLADM_STATUS_OK) {
2847                    goto done;
2848            }

2850            if (!(state->ls_flags & flags)) {
2851                    status = DLADM_STATUS_NOTFOUND;
2852                    goto done;
2853            }

2855            if (state->ls_flags == DLADM_OPT_ACTIVE) {
2856                    dladm_attr_t    dlattr;

2858                    if (class == DATALINK_CLASS_PHYS) {
2859                            dladm_phys_attr_t       dpa;
2860                            dlpi_handle_t           dh;
2861                            dlpi_info_t             dlinfo;

2863                            if ((status = dladm_phys_info(handle, linkid, &dpa,
2864                                DLADM_OPT_ACTIVE)) != DLADM_STATUS_OK) {
2865                                    goto done;
2866                            }

2868                            if (!dpa.dp_novanity)
2869                                    goto link_mtu;

2871                            /*
2872                             * This is a physical link that does not have
2873                             * vanity naming support.
2874                             */
2875                            if (dlpi_open(dpa.dp_dev, &dh, DLPI_DEVONLY) !=
2876                                DLPI_SUCCESS) {
2877                                    status = DLADM_STATUS_NOTFOUND;
2878                                    goto done;
2879                            }

2881                            if (dlpi_info(dh, &dlinfo, 0) != DLPI_SUCCESS) {
2882                                    dlpi_close(dh);
2883                                    status = DLADM_STATUS_BADARG;
2884                                    goto done;
2885                            }

2887                            dlpi_close(dh);
2888                            mtu = dlinfo.di_max_sdu;
2889                    } else {
2890 link_mtu:
2891                            status = dladm_info(handle, linkid, &dlattr);
2892                            if (status != DLADM_STATUS_OK)
2893                                    goto done;
2894                            mtu = dlattr.da_max_sdu;
2895                    }
2896            }

2898            (void) snprintf(lbuf->link_name, sizeof (lbuf->link_name),
2899                "%s", link);
2900            (void) dladm_class2str(class, lbuf->link_class);
2901            if (state->ls_flags == DLADM_OPT_ACTIVE) {
2902                    (void) snprintf(lbuf->link_mtu, sizeof (lbuf->link_mtu),
2903                        "%u", mtu);
2904                    (void) get_linkstate(link, B_TRUE, lbuf->link_state);
2905            }

2907            print_link_topology(state, linkid, class, lbuf);
2908 done:
2909            return (status);
2910 }
```

```
2912 /* ARGSUSED */
2913 static int
2914 show_link(dladm_handle_t dh, datalink_id_t linkid, void *arg)
2915 {
2916         show_state_t            *state = (show_state_t *)arg;
2917         dladm_status_t          status;
2918         link_fields_buf_t       lbuf;

2920         /*
2921          * first get all the link attributes into lbuf;
2922          */
2923         bzero(&lbuf, sizeof (link_fields_buf_t));
2924         if ((status = print_link(state, linkid, &lbuf)) == DLADM_STATUS_OK)
2925                 ofmt_print(state->ls_ofmt, &lbuf);
2926         state->ls_status = status;
2927         return (DLADM_WALK_CONTINUE);
2928 }

2930 static boolean_t
2931 print_link_stats_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
2932 {
2933         link_args_t *largs = ofarg->ofmt_cbarg;
2934         pktsum_t *diff_stats = largs->link_s_psum;

2936         switch (ofarg->ofmt_id) {
2937         case LINK_S_LINK:
2938                 (void) snprintf(buf, bufsize, "%s", largs->link_s_link);
2939                 break;
2940         case LINK_S_IPKTS:
2941                 (void) snprintf(buf, bufsize, "%llu", diff_stats->ipackets);
2942                 break;
2943         case LINK_S_RBYTES:
2944                 (void) snprintf(buf, bufsize, "%llu", diff_stats->rbytes);
2945                 break;
2946         case LINK_S_IERRORS:
2947                 (void) snprintf(buf, bufsize, "%u", diff_stats->ierrors);
2948                 break;
2949         case LINK_S_OPKTS:
2950                 (void) snprintf(buf, bufsize, "%llu", diff_stats->opackets);
2951                 break;
2952         case LINK_S_OBYTES:
2953                 (void) snprintf(buf, bufsize, "%llu", diff_stats->obytes);
2954                 break;
2955         case LINK_S_OERRORS:
2956                 (void) snprintf(buf, bufsize, "%u", diff_stats->oerrors);
2957                 break;
2958         default:
2959                 die("invalid input");
2960                 break;
2961         }
2962         return (B_TRUE);
2963 }

2965 static int
2966 show_link_stats(dladm_handle_t dh, datalink_id_t linkid, void *arg)
2967 {
2968         char                    link[DLPI_LINKNAME_MAX];
2969         datalink_class_t        class;
2970         show_state_t            *state = arg;
2971         pktsum_t                stats, diff_stats;
2972         dladm_phys_attr_t       dpa;
2973         link_args_t             largs;

2975         if (state->ls_firstonly) {
2976                 if (state->ls_donefirst)
```



```
2977                         return (DLADM_WALK_CONTINUE);
2978                 state->ls_donefirst = B_TRUE;
2979         } else {
2980                 bzero(&state->ls_prevstats, sizeof (state->ls_prevstats));
2981         }

2983         if (dladm_datalink_id2info(dh, linkid, NULL, &class, NULL, link,
2984             DLPI_LINKNAME_MAX) != DLADM_STATUS_OK) {
2985                 return (DLADM_WALK_CONTINUE);
2986         }

2988         if (class == DATALINK_CLASS_PHYS) {
2989                 if (dladm_phys_info(dh, linkid, &dpa, DLADM_OPT_ACTIVE) !=
2990                     DLADM_STATUS_OK) {
2991                         return (DLADM_WALK_CONTINUE);
2992                 }
2993                 if (dpa.dp_novanity)
2994                         get_mac_stats(dpa.dp_dev, &stats);
2995                 else
2996                         get_link_stats(link, &stats);
2997         } else {
2998                 get_link_stats(link, &stats);
2999         }
3000         dladm_stats_diff(&diff_stats, &stats, &state->ls_prevstats);

3002         largs.link_s_link = link;
3003         largs.link_s_psum = &diff_stats;
3004         ofmt_print(state->ls_ofmt, &largs);

3006         state->ls_prevstats = stats;
3007         return (DLADM_WALK_CONTINUE);
3008 }


3011 static dladm_status_t
3012 print_aggr_info(show_grp_state_t *state, const char *link,
3013     dladm_aggr_grp_attr_t *ginfop)
3014 {
3015         char                    addr_str[ETHERADDRL * 3];
3016         laggr_fields_buf_t      lbuf;

3018         (void) snprintf(lbuf.laggr_name, sizeof (lbuf.laggr_name),
3019             "%s", link);

3021         (void) dladm_aggr_policy2str(ginfop->lg_policy,
3022             lbuf.laggr_policy);

3024         if (ginfop->lg_mac_fixed) {
3025                 (void) dladm_aggr_macaddr2str(ginfop->lg_mac, addr_str);
3026                 (void) snprintf(lbuf.laggr_addrpolicy,
3027                     sizeof (lbuf.laggr_addrpolicy), "fixed (%s)", addr_str);
3028         } else {
3029                 (void) snprintf(lbuf.laggr_addrpolicy,
3030                     sizeof (lbuf.laggr_addrpolicy), "auto");
3031         }

3033         (void) dladm_aggr_lacpmode2str(ginfop->lg_lacp_mode,
3034             lbuf.laggr_lacpactivity);
3035         (void) dladm_aggr_lacptimer2str(ginfop->lg_lacp_timer,
3036             lbuf.laggr_lacptimer);
3037         (void) snprintf(lbuf.laggr_flags, sizeof (lbuf.laggr_flags), "%c----",
3038             ginfop->lg_force ? 'f' : '-');

3040         ofmt_print(state->gs_ofmt, &lbuf);

3042         return (DLADM_STATUS_OK);
```

```
}

static boolean_t
print_xaggr_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
{
        const laggr_args_t      *l = ofarg->ofmt_cbarg;
        boolean_t               is_port = (l->laggr_lport >= 0);
        char                    tmpbuf[DLADM_STRSIZE];
        const char              *objname;
        dladm_aggr_port_attr_t  *portp;
        dladm_phys_attr_t       dpa;

        if (is_port) {
                portp = &(l->laggr_ginfop->lg_ports[l->laggr_lport]);
                if (dladm_phys_info(handle, portp->lp_linkid, &dpa,
                    DLADM_OPT_ACTIVE) != DLADM_STATUS_OK)
                        objname = "?";
                else
                        objname = dpa.dp_dev;
        } else {
                objname = l->laggr_link;
        }

        switch (ofarg->ofmt_id) {
        case AGGR_X_LINK:
                (void) snprintf(buf, bufsize, "%s",
                    (is_port && !l->laggr_parsable ? " " : l->laggr_link));
                break;
        case AGGR_X_PORT:
                if (is_port) {
                        if (dladm_datalink_id2info(handle, portp->lp_linkid,
                            NULL, NULL, NULL, buf, bufsize) != DLADM_STATUS_OK)
                                (void) sprintf(buf, "?");
                }
                break;

        case AGGR_X_SPEED:
                (void) snprintf(buf, bufsize, "%uMb",
                    (uint_t)((get_ifspeed(objname, !is_port)) / 1000000ull));
                break;

        case AGGR_X_DUPLEX:
                (void) get_linkduplex(objname, !is_port, tmpbuf);
                (void) strlcpy(buf, tmpbuf, bufsize);
                break;

        case AGGR_X_STATE:
                (void) get_linkstate(objname, !is_port, tmpbuf);
                (void) strlcpy(buf, tmpbuf, bufsize);
                break;
        case AGGR_X_ADDRESS:
                (void) dladm_aggr_macaddr2str(
                    (is_port ? portp->lp_mac : l->laggr_ginfop->lg_mac),
                    tmpbuf);
                (void) strlcpy(buf, tmpbuf, bufsize);
                break;
        case AGGR_X_PORTSTATE:
                if (is_port) {
                        (void) dladm_aggr_portstate2str(portp->lp_state,
                            tmpbuf);
                        (void) strlcpy(buf, tmpbuf, bufsize);
                }
                break;
        }
err:
        *(l->laggr_status) = DLADM_STATUS_OK;
```



```
        return (B_TRUE);
}

static dladm_status_t
print_aggr_extended(show_grp_state_t *state, const char *link,
    dladm_aggr_grp_attr_t *ginfop)
{
        int                     i;
        dladm_status_t          status;
        laggr_args_t            largs;

        largs.laggr_lport = -1;
        largs.laggr_link = link;
        largs.laggr_ginfop = ginfop;
        largs.laggr_status = &status;
        largs.laggr_parsable = state->gs_parsable;

        ofmt_print(state->gs_ofmt, &largs);

        if (status != DLADM_STATUS_OK)
                goto done;

        for (i = 0; i < ginfop->lg_nports; i++) {
                largs.laggr_lport = i;
                ofmt_print(state->gs_ofmt, &largs);
                if (status != DLADM_STATUS_OK)
                        goto done;
        }

        status = DLADM_STATUS_OK;
done:
        return (status);
}

static boolean_t
print_lacp_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
{
        const laggr_args_t      *l = ofarg->ofmt_cbarg;
        int                     portnum;
        boolean_t               is_port = (l->laggr_lport >= 0);
        dladm_aggr_port_attr_t  *portp;
        aggr_lacp_state_t       *lstate;

        if (!is_port)
                return (B_FALSE); /* cannot happen! */

        portnum = l->laggr_lport;
        portp = &(l->laggr_ginfop->lg_ports[portnum]);
        lstate = &(portp->lp_lacp_state);

        switch (ofarg->ofmt_id) {
        case AGGR_L_LINK:
                (void) snprintf(buf, bufsize, "%s",
                    (portnum > 0 ? "" : l->laggr_link));
                break;

        case AGGR_L_PORT:
                if (dladm_datalink_id2info(handle, portp->lp_linkid, NULL, NULL,
                    NULL, buf, bufsize) != DLADM_STATUS_OK)
                        (void) sprintf(buf, "?");
                break;

        case AGGR_L_AGGREGATABLE:
                (void) snprintf(buf, bufsize, "%s",
                    (lstate->bit.aggregation ? "yes" : "no"));
                break;
```

```
3176         case AGGR_L_SYNC:
3177                 (void) snprintf(buf, bufsize, "%s",
3178                     (lstate->bit.sync ? "yes" : "no"));
3179                 break;

3181         case AGGR_L_COLL:
3182                 (void) snprintf(buf, bufsize, "%s",
3183                     (lstate->bit.collecting ? "yes" : "no"));
3184                 break;

3186         case AGGR_L_DIST:
3187                 (void) snprintf(buf, bufsize, "%s",
3188                     (lstate->bit.distributing ? "yes" : "no"));
3189                 break;

3191         case AGGR_L_DEFAULTED:
3192                 (void) snprintf(buf, bufsize, "%s",
3193                     (lstate->bit.defaulted ? "yes" : "no"));
3194                 break;

3196         case AGGR_L_EXPIRED:
3197                 (void) snprintf(buf, bufsize, "%s",
3198                     (lstate->bit.expired ? "yes" : "no"));
3199                 break;
3200         }

3202         *(l->laggr_status) = DLADM_STATUS_OK;
3203         return (B_TRUE);
3204 }

3206 static dladm_status_t
3207 print_aggr_lacp(show_grp_state_t *state, const char *link,
3208     dladm_aggr_grp_attr_t *ginfop)
3209 {
3210         int             i;
3211         dladm_status_t  status;
3212         laggr_args_t    largs;

3214         largs.laggr_link = link;
3215         largs.laggr_ginfop = ginfop;
3216         largs.laggr_status = &status;

3218         for (i = 0; i < ginfop->lg_nports; i++) {
3219                 largs.laggr_lport = i;
3220                 ofmt_print(state->gs_ofmt, &largs);
3221                 if (status != DLADM_STATUS_OK)
3222                         goto done;
3223         }

3225         status = DLADM_STATUS_OK;
3226 done:
3227         return (status);
3228 }

3230 static boolean_t
3231 print_aggr_stats_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
3232 {
3233         const laggr_args_t      *l = ofarg->ofmt_cbarg;
3234         int                     portnum;
3235         boolean_t               is_port = (l->laggr_lport >= 0);
3236         dladm_aggr_port_attr_t  *portp;
3237         dladm_status_t          *stat, status;
3238         pktsum_t                *diff_stats;

3240         stat = l->laggr_status;
```



```
3241         *stat = DLADM_STATUS_OK;

3243         if (is_port) {
3244                 portnum = l->laggr_lport;
3245                 portp = &(l->laggr_ginfop->lg_ports[portnum]);

3247                 if ((status = dladm_datalink_id2info(handle,
3248                     portp->lp_linkid, NULL, NULL, NULL, buf, bufsize)) !=
3249                     DLADM_STATUS_OK) {
3250                         goto err;
3251                 }
3252                 diff_stats = l->laggr_diffstats;
3253         }

3255         switch (ofarg->ofmt_id) {
3256         case AGGR_S_LINK:
3257                 (void) snprintf(buf, bufsize, "%s",
3258                     (is_port ? "" : l->laggr_link));
3259                 break;
3260         case AGGR_S_PORT:
3261                 /*
3262                  * if (is_port), buf has port name. Otherwise we print
3263                  * STR_UNDEF_VAL
3264                  */
3265                 break;

3267         case AGGR_S_IPKTS:
3268                 if (is_port) {
3269                         (void) snprintf(buf, bufsize, "%llu",
3270                             diff_stats->ipackets);
3271                 } else {
3272                         (void) snprintf(buf, bufsize, "%llu",
3273                             l->laggr_pktsumtot->ipackets);
3274                 }
3275                 break;

3277         case AGGR_S_RBYTES:
3278                 if (is_port) {
3279                         (void) snprintf(buf, bufsize, "%llu",
3280                             diff_stats->rbytes);
3281                 } else {
3282                         (void) snprintf(buf, bufsize, "%llu",
3283                             l->laggr_pktsumtot->rbytes);
3284                 }
3285                 break;

3287         case AGGR_S_OPKTS:
3288                 if (is_port) {
3289                         (void) snprintf(buf, bufsize, "%llu",
3290                             diff_stats->opackets);
3291                 } else {
3292                         (void) snprintf(buf, bufsize, "%llu",
3293                             l->laggr_pktsumtot->opackets);
3294                 }
3295                 break;
3296         case AGGR_S_OBYTES:
3297                 if (is_port) {
3298                         (void) snprintf(buf, bufsize, "%llu",
3299                             diff_stats->obytes);
3300                 } else {
3301                         (void) snprintf(buf, bufsize, "%llu",
3302                             l->laggr_pktsumtot->obytes);
3303                 }
3304                 break;

3306         case AGGR_S_IPKTDIST:
```

```
3307                 if (is_port) {
3308                         (void) snprintf(buf, bufsize, "%-6.1f",
3309                             (double)diff_stats->ipackets/
3310                             (double)l->laggr_pktsumtot->ipackets * 100);
3311                 }
3312                 break;
3313         case AGGR_S_OPKTDIST:
3314                 if (is_port) {
3315                         (void) snprintf(buf, bufsize, "%-6.1f",
3316                             (double)diff_stats->opackets/
3317                             (double)l->laggr_pktsumtot->opackets * 100);
3318                 }
3319                 break;
3320         }
3321         return (B_TRUE);

3323 err:
3324         *stat = status;
3325         return (B_TRUE);
3326 }

3328 static dladm_status_t
3329 print_aggr_stats(show_grp_state_t *state, const char *link,
3330     dladm_aggr_grp_attr_t *ginfop)
3331 {
3332         dladm_phys_attr_t       dpa;
3333         dladm_aggr_port_attr_t  *portp;
3334         pktsum_t                pktsumtot, *port_stat;
3335         dladm_status_t          status;
3336         int                     i;
3337         laggr_args_t            largs;

3339         /* sum the ports statistics */
3340         bzero(&pktsumtot, sizeof (pktsumtot));

3342         /* Allocate memory to keep stats of each port */
3343         port_stat = malloc(ginfop->lg_nports * sizeof (pktsum_t));
3344         if (port_stat == NULL) {
3345                 /* Bail out; no memory */
3346                 return (DLADM_STATUS_NOMEM);
3347         }


3350         for (i = 0; i < ginfop->lg_nports; i++) {

3352                 portp = &(ginfop->lg_ports[i]);
3353                 if ((status = dladm_phys_info(handle, portp->lp_linkid, &dpa,
3354                     DLADM_OPT_ACTIVE)) != DLADM_STATUS_OK) {
3355                         goto done;
3356                 }

3358                 get_mac_stats(dpa.dp_dev, &port_stat[i]);

3360                 /*
3361                  * Let's re-use gs_prevstats[] to store the difference of the
3362                  * counters since last use. We will store the new stats from
3363                  * port_stat[] once we have the stats displayed.
3364                  */

3366                 dladm_stats_diff(&state->gs_prevstats[i], &port_stat[i],
3367                     &state->gs_prevstats[i]);
3368                 dladm_stats_total(&pktsumtot, &pktsumtot,
3369                     &state->gs_prevstats[i]);
3370         }

3372         largs.laggr_lport = -1;
```



```
3373         largs.laggr_link = link;
3374         largs.laggr_ginfop = ginfop;
3375         largs.laggr_status = &status;
3376         largs.laggr_pktsumtot = &pktsumtot;

3378         ofmt_print(state->gs_ofmt, &largs);

3380         if (status != DLADM_STATUS_OK)
3381                 goto done;

3383         for (i = 0; i < ginfop->lg_nports; i++) {
3384                 largs.laggr_lport = i;
3385                 largs.laggr_diffstats = &state->gs_prevstats[i];
3386                 ofmt_print(state->gs_ofmt, &largs);
3387                 if (status != DLADM_STATUS_OK)
3388                         goto done;
3389         }

3391         status = DLADM_STATUS_OK;
3392         for (i = 0; i < ginfop->lg_nports; i++)
3393                 state->gs_prevstats[i] = port_stat[i];

3395 done:
3396         free(port_stat);
3397         return (status);
3398 }

3400 static dladm_status_t
3401 print_aggr(show_grp_state_t *state, datalink_id_t linkid)
3402 {
3403         char                    link[MAXLINKNAMELEN];
3404         dladm_aggr_grp_attr_t   ginfo;
3405         uint32_t                flags;
3406         dladm_status_t          status;

3408         bzero(&ginfo, sizeof (dladm_aggr_grp_attr_t));
3409         if ((status = dladm_datalink_id2info(handle, linkid, &flags, NULL,
3410             NULL, link, MAXLINKNAMELEN)) != DLADM_STATUS_OK) {
3411                 return (status);
3412         }

3414         if (!(state->gs_flags & flags))
3415                 return (DLADM_STATUS_NOTFOUND);

3417         status = dladm_aggr_info(handle, linkid, &ginfo, state->gs_flags);
3418         if (status != DLADM_STATUS_OK)
3419                 return (status);

3421         if (state->gs_lacp)
3422                 status = print_aggr_lacp(state, link, &ginfo);
3423         else if (state->gs_extended)
3424                 status = print_aggr_extended(state, link, &ginfo);
3425         else if (state->gs_stats)
3426                 status = print_aggr_stats(state, link, &ginfo);
3427         else
3428                 status = print_aggr_info(state, link, &ginfo);

3430 done:
3431         free(ginfo.lg_ports);
3432         return (status);
3433 }

3435 /* ARGSUSED */
3436 static int
3437 show_aggr(dladm_handle_t dh, datalink_id_t linkid, void *arg)
3438 {
```

```
3439         show_grp_state_t        *state = arg;

3441         state->gs_status = print_aggr(state, linkid);
3442         return (DLADM_WALK_CONTINUE);
3443 }

3445 static void
3446 do_show_link(int argc, char *argv[], const char *use)
3447 {
3448         int             option;
3449         boolean_t       s_arg = B_FALSE;
3450         boolean_t       S_arg = B_FALSE;
3451         boolean_t       i_arg = B_FALSE;
3452         uint32_t        flags = DLADM_OPT_ACTIVE;
3453         boolean_t       p_arg = B_FALSE;
3454         datalink_id_t   linkid = DATALINK_ALL_LINKID;
3455         char            linkname[MAXLINKNAMELEN];
3456         uint32_t        interval = 0;
3457         show_state_t    state;
3458         dladm_status_t  status;
3459         boolean_t       o_arg = B_FALSE;
3460         char            *fields_str = NULL;
3461         char            *all_active_fields = "link,class,mtu,state,bridge,over";
3462         char            *all_inactive_fields = "link,class,bridge,over";
3463         char            *allstat_fields =
3464             "link,ipackets,rbytes,ierrors,opackets,obytes,oerrors";
3465         ofmt_handle_t   ofmt;
3466         ofmt_status_t   oferr;
3467         uint_t          ofmtflags = 0;

3469         bzero(&state, sizeof (state));

3471         opterr = 0;
3472         while ((option = getopt_long(argc, argv, ":pPsSi:o:",
3473             show_lopts, NULL)) != -1) {
3474                 switch (option) {
3475                 case 'p':
3476                         if (p_arg)
3477                                 die_optdup(option);

3479                         p_arg = B_TRUE;
3480                         break;
3481                 case 's':
3482                         if (s_arg)
3483                                 die_optdup(option);

3485                         s_arg = B_TRUE;
3486                         break;
3487                 case 'P':
3488                         if (flags != DLADM_OPT_ACTIVE)
3489                                 die_optdup(option);

3491                         flags = DLADM_OPT_PERSIST;
3492                         break;
3493                 case 'S':
3494                         if (S_arg)
3495                                 die_optdup(option);

3497                         S_arg = B_TRUE;
3498                         break;
3499                 case 'o':
3500                         o_arg = B_TRUE;
3501                         fields_str = optarg;
3502                         break;
3503                 case 'i':
3504                         if (i_arg)
```



```
3505                                 die_optdup(option);

3507                         i_arg = B_TRUE;
3508                         if (!dladm_str2interval(optarg, &interval))
3509                                 die("invalid interval value '%s'", optarg);
3510                         break;
3511                 default:
3512                         die_opterr(optopt, option, use);
3513                         break;
3514                 }
3515         }

3517         if (i_arg && !(s_arg || S_arg))
3518                 die("the option -i can be used only with -s or -S");

3520         if (s_arg && S_arg)
3521                 die("the -s option cannot be used with -S");

3523         if (s_arg && flags != DLADM_OPT_ACTIVE)
3524                 die("the option -P cannot be used with -s");

3526         if (S_arg && (p_arg || flags != DLADM_OPT_ACTIVE))
3527                 die("the option -%c cannot be used with -S", p_arg ? 'p' : 'P');

3529         /* get link name (optional last argument) */
3530         if (optind == (argc-1)) {
3531                 uint32_t        f;

3533                 if (strlcpy(linkname, argv[optind], MAXLINKNAMELEN) >=
3534                     MAXLINKNAMELEN)
3535                         die("link name too long");
3536                 if ((status = dladm_name2info(handle, linkname, &linkid, &f,
3537                     NULL, NULL)) != DLADM_STATUS_OK) {
3538                         die_dlerr(status, "link %s is not valid", linkname);
3539                 }

3541                 if (!(f & flags)) {
3542                         die_dlerr(DLADM_STATUS_BADARG, "link %s is %s",
3543                             argv[optind], flags == DLADM_OPT_PERSIST ?
3544                             "a temporary link" : "temporarily removed");
3545                 }
3546         } else if (optind != argc) {
3547                 usage();
3548         }

3550         if (p_arg && !o_arg)
3551                 die("-p requires -o");

3553         if (S_arg) {
3554                 dladm_continuous(handle, linkid, NULL, interval, LINK_REPORT);
3555                 return;
3556         }

3558         if (p_arg && strcasecmp(fields_str, "all") == 0)
3559                 die("\"-o all\" is invalid with -p");

3561         if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0)) {
3562                 if (s_arg)
3563                         fields_str = allstat_fields;
3564                 else if (flags & DLADM_OPT_ACTIVE)
3565                         fields_str = all_active_fields;
3566                 else
3567                         fields_str = all_inactive_fields;
3568         }

3570         state.ls_parsable = p_arg;
```

```
3571         state.ls_flags = flags;
3572         state.ls_donefirst = B_FALSE;

3574         if (s_arg) {
3575                 link_stats(linkid, interval, fields_str, &state);
3576                 return;
3577         }
3578         if (state.ls_parsable)
3579                 ofmtflags |= OFMT_PARSABLE;
3580         oferr = ofmt_open(fields_str, link_fields, ofmtflags, 0, &ofmt);
3581         dladm_ofmt_check(oferr, state.ls_parsable, ofmt);
3582         state.ls_ofmt = ofmt;

3584         if (linkid == DATALINK_ALL_LINKID) {
3585                 (void) dladm_walk_datalink_id(show_link, handle, &state,
3586                     DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE, flags);
3587         } else {
3588                 (void) show_link(handle, linkid, &state);
3589                 if (state.ls_status != DLADM_STATUS_OK) {
3590                         die_dlerr(state.ls_status, "failed to show link %s",
3591                             argv[optind]);
3592                 }
3593         }
3594         ofmt_close(ofmt);
3595 }

3597 static void
3598 do_show_aggr(int argc, char *argv[], const char *use)
3599 {
3600         boolean_t               L_arg = B_FALSE;
3601         boolean_t               s_arg = B_FALSE;
3602         boolean_t               i_arg = B_FALSE;
3603         boolean_t               p_arg = B_FALSE;
3604         boolean_t               x_arg = B_FALSE;
3605         show_grp_state_t        state;
3606         uint32_t                flags = DLADM_OPT_ACTIVE;
3607         datalink_id_t           linkid = DATALINK_ALL_LINKID;
3608         int                     option;
3609         uint32_t                interval = 0;
3610         int                     key;
3611         dladm_status_t          status;
3612         boolean_t               o_arg = B_FALSE;
3613         char                    *fields_str = NULL;
3614         char                    *all_fields =
3615             "link,policy,addrpolicy,lacpactivity,lacptimer,flags";
3616         char                    *all_lacp_fields =
3617             "link,port,aggregatable,sync,coll,dist,defaulted,expired";
3618         char                    *all_stats_fields =
3619             "link,port,ipackets,rbytes,opackets,obytes,ipktdist,opktdist";
3620         char                    *all_extended_fields =
3621             "link,port,speed,duplex,state,address,portstate";
3622         const ofmt_field_t      *pf;
3623         ofmt_handle_t           ofmt;
3624         ofmt_status_t           oferr;
3625         uint_t                  ofmtflags = 0;

3627         opterr = 0;
3628         while ((option = getopt_long(argc, argv, ":LpPxsi:o:",
3629             show_lopts, NULL)) != -1) {
3630                 switch (option) {
3631                 case 'L':
3632                         if (L_arg)
3633                                 die_optdup(option);

3635                         L_arg = B_TRUE;
3636                         break;
```



```
3637                 case 'p':
3638                         if (p_arg)
3639                                 die_optdup(option);

3641                         p_arg = B_TRUE;
3642                         break;
3643                 case 'x':
3644                         if (x_arg)
3645                                 die_optdup(option);

3647                         x_arg = B_TRUE;
3648                         break;
3649                 case 'P':
3650                         if (flags != DLADM_OPT_ACTIVE)
3651                                 die_optdup(option);

3653                         flags = DLADM_OPT_PERSIST;
3654                         break;
3655                 case 's':
3656                         if (s_arg)
3657                                 die_optdup(option);

3659                         s_arg = B_TRUE;
3660                         break;
3661                 case 'o':
3662                         o_arg = B_TRUE;
3663                         fields_str = optarg;
3664                         break;
3665                 case 'i':
3666                         if (i_arg)
3667                                 die_optdup(option);

3669                         i_arg = B_TRUE;
3670                         if (!dladm_str2interval(optarg, &interval))
3671                                 die("invalid interval value '%s'", optarg);
3672                         break;
3673                 default:
3674                         die_opterr(optopt, option, use);
3675                         break;
3676                 }
3677         }

3679         if (p_arg && !o_arg)
3680                 die("-p requires -o");

3682         if (p_arg && strcasecmp(fields_str, "all") == 0)
3683                 die("\"-o all\" is invalid with -p");

3685         if (i_arg && !s_arg)
3686                 die("the option -i can be used only with -s");

3688         if (s_arg && (L_arg || p_arg || x_arg || flags != DLADM_OPT_ACTIVE)) {
3689                 die("the option -%c cannot be used with -s",
3690                     L_arg ? 'L' : (p_arg ? 'p' : (x_arg ? 'x' : 'P')));
3691         }

3693         if (L_arg && flags != DLADM_OPT_ACTIVE)
3694                 die("the option -P cannot be used with -L");

3696         if (x_arg && (L_arg || flags != DLADM_OPT_ACTIVE))
3697                 die("the option -%c cannot be used with -x", L_arg ? 'L' : 'P');

3699         /* get aggregation key or aggrname (optional last argument) */
3700         if (optind == (argc-1)) {
3701                 if (!str2int(argv[optind], &key)) {
3702                         status = dladm_name2info(handle, argv[optind],
```

```
3703                            &linkid, NULL, NULL, NULL);
3704                } else {
3705                        status = dladm_key2linkid(handle, (uint16_t)key,
3706                            &linkid, DLADM_OPT_ACTIVE);
3707                }

3709                if (status != DLADM_STATUS_OK)
3710                        die("non-existent aggregation '%s'", argv[optind]);

3712        } else if (optind != argc) {
3713                usage();
3714        }

3716        bzero(&state, sizeof (state));
3717        state.gs_lacp = L_arg;
3718        state.gs_stats = s_arg;
3719        state.gs_flags = flags;
3720        state.gs_parsable = p_arg;
3721        state.gs_extended = x_arg;

3723        if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0)) {
3724                if (state.gs_lacp)
3725                        fields_str = all_lacp_fields;
3726                else if (state.gs_stats)
3727                        fields_str = all_stats_fields;
3728                else if (state.gs_extended)
3729                        fields_str = all_extended_fields;
3730                else
3731                        fields_str = all_fields;
3732        }

3734        if (state.gs_lacp) {
3735                pf = aggr_l_fields;
3736        } else if (state.gs_stats) {
3737                pf = aggr_s_fields;
3738        } else if (state.gs_extended) {
3739                pf = aggr_x_fields;
3740        } else {
3741                pf = laggr_fields;
3742        }

3744        if (state.gs_parsable)
3745                ofmtflags |= OFMT_PARSABLE;
3746        oferr = ofmt_open(fields_str, pf, ofmtflags, 0, &ofmt);
3747        dladm_ofmt_check(oferr, state.gs_parsable, ofmt);
3748        state.gs_ofmt = ofmt;

3750        if (s_arg) {
3751                aggr_stats(linkid, &state, interval);
3752                ofmt_close(ofmt);
3753                return;
3754        }

3756        if (linkid == DATALINK_ALL_LINKID) {
3757                (void) dladm_walk_datalink_id(show_aggr, handle, &state,
3758                    DATALINK_CLASS_AGGR, DATALINK_ANY_MEDIATYPE, flags);
3759        } else {
3760                (void) show_aggr(handle, linkid, &state);
3761                if (state.gs_status != DLADM_STATUS_OK) {
3762                        die_dlerr(state.gs_status, "failed to show aggr %s",
3763                            argv[optind]);
3764                }
3765        }
3766        ofmt_close(ofmt);
3767 }
```



```
3769 static dladm_status_t
3770 print_phys_default(show_state_t *state, datalink_id_t linkid,
3771     const char *link, uint32_t flags, uint32_t media)
3772 {
3773        dladm_phys_attr_t dpa;
3774        dladm_status_t status;
3775        link_fields_buf_t pattr;

3777        status = dladm_phys_info(handle, linkid, &dpa, state->ls_flags);
3778        if (status != DLADM_STATUS_OK)
3779                goto done;

3781        (void) snprintf(pattr.link_phys_device,
3782            sizeof (pattr.link_phys_device), "%s", dpa.dp_dev);
3783        (void) dladm_media2str(media, pattr.link_phys_media);
3784        if (state->ls_flags == DLADM_OPT_ACTIVE) {
3785                boolean_t       islink;

3787                if (!dpa.dp_novanity) {
3788                        (void) strlcpy(pattr.link_name, link,
3789                            sizeof (pattr.link_name));
3790                        islink = B_TRUE;
3791                } else {
3792                        /*
3793                         * This is a physical link that does not have
3794                         * vanity naming support.
3795                         */
3796                        (void) strlcpy(pattr.link_name, dpa.dp_dev,
3797                            sizeof (pattr.link_name));
3798                        islink = B_FALSE;
3799                }

3801                (void) get_linkstate(pattr.link_name, islink,
3802                    pattr.link_phys_state);
3803                (void) snprintf(pattr.link_phys_speed,
3804                    sizeof (pattr.link_phys_speed), "%u",
3805                    (uint_t)((get_ifspeed(pattr.link_name,
3806                    islink)) / 1000000ull));
3807                (void) get_linkduplex(pattr.link_name, islink,
3808                    pattr.link_phys_duplex);
3809        } else {
3810                (void) snprintf(pattr.link_name, sizeof (pattr.link_name),
3811                    "%s", link);
3812                (void) snprintf(pattr.link_flags, sizeof (pattr.link_flags),
3813                    "%c----", flags & DLADM_OPT_ACTIVE ? '-' : 'r');
3814        }

3816        ofmt_print(state->ls_ofmt, &pattr);

3818 done:
3819        return (status);
3820 }

3822 typedef struct {
3823        show_state_t    *ms_state;
3824        char            *ms_link;
3825        dladm_macaddr_attr_t *ms_mac_attr;
3826 } print_phys_mac_state_t;

3828 /*
3829  *  callback for ofmt_print()
3830  */
3831 static boolean_t
3832 print_phys_one_mac_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
3833 {
3834        print_phys_mac_state_t *mac_state = ofarg->ofmt_cbarg;
```

```
3835         dladm_macaddr_attr_t *attr = mac_state->ms_mac_attr;
3836         boolean_t is_primary = (attr->ma_slot == 0);
3837         boolean_t is_parsable = mac_state->ms_state->ls_parsable;

3839         switch (ofarg->ofmt_id) {
3840         case PHYS_M_LINK:
3841                 (void) snprintf(buf, bufsize, "%s",
3842                     (is_primary || is_parsable) ? mac_state->ms_link : " ");
3843                 break;
3844         case PHYS_M_SLOT:
3845                 if (is_primary)
3846                         (void) snprintf(buf, bufsize, gettext("primary"));
3847                 else
3848                         (void) snprintf(buf, bufsize, "%d", attr->ma_slot);
3849                 break;
3850         case PHYS_M_ADDRESS:
3851                 (void) dladm_aggr_macaddr2str(attr->ma_addr, buf);
3852                 break;
3853         case PHYS_M_INUSE:
3854                 (void) snprintf(buf, bufsize, "%s",
3855                     attr->ma_flags & DLADM_MACADDR_USED ? gettext("yes") :
3856                     gettext("no"));
3857                 break;
3858         case PHYS_M_CLIENT:
3859                 /*
3860                  * CR 6678526: resolve link id to actual link name if
3861                  * it is valid.
3862                  */
3863                 (void) snprintf(buf, bufsize, "%s", attr->ma_client_name);
3864                 break;
3865         }

3867         return (B_TRUE);
3868 }

3870 typedef struct {
3871         show_state_t    *hs_state;
3872         char            *hs_link;
3873         dladm_hwgrp_attr_t *hs_grp_attr;
3874 } print_phys_hwgrp_state_t;

3876 static boolean_t
3877 print_phys_one_hwgrp_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
3878 {
3879         int             i;
3880         boolean_t       first = B_TRUE;
3881         int             start = -1;
3882         int             end = -1;
3883         char            ringstr[RINGSTRLEN];
3884         char            ringsubstr[RINGSTRLEN];

3886         print_phys_hwgrp_state_t *hg_state = ofarg->ofmt_cbarg;
3887         dladm_hwgrp_attr_t *attr = hg_state->hs_grp_attr;

3889         switch (ofarg->ofmt_id) {
3890         case PHYS_H_LINK:
3891                 (void) snprintf(buf, bufsize, "%s", attr->hg_link_name);
3892                 break;
3893         case PHYS_H_RINGTYPE:
3894                 (void) snprintf(buf, bufsize, "%s",
3895                     attr->hg_grp_type == DLADM_HWGRP_TYPE_RX ? "RX" : "TX");
3896                 break;
3897         case PHYS_H_RINGS:
3898                 ringstr[0] = '\0';
3899                 for (i = 0; i < attr->hg_n_rings; i++) {
3900                         uint_t  index = attr->hg_rings[i];
```



```
3902                         if (start == -1) {
3903                                 start = index;
3904                                 end = index;
3905                         } else if (index == end + 1) {
3906                                 end = index;
3907                         } else {
3908                                 if (start == end) {
3909                                         if (first) {
3910                                                 (void) snprintf(
3911                                                     ringsubstr,
3912                                                     RINGSTRLEN, "%d",
3913                                                     start);
3914                                                 first = B_FALSE;
3915                                         } else {
3916                                                 (void) snprintf(
3917                                                     ringsubstr,
3918                                                     RINGSTRLEN, ",%d",
3919                                                     start);
3920                                         }
3921                                 } else {
3922                                         if (first) {
3923                                                 (void) snprintf(
3924                                                     ringsubstr,
3925                                                     RINGSTRLEN,
3926                                                     "%d-%d",
3927                                                     start, end);
3928                                                 first = B_FALSE;
3929                                         } else {
3930                                                 (void) snprintf(
3931                                                     ringsubstr,
3932                                                     RINGSTRLEN,
3933                                                     ",%d-%d",
3934                                                     start, end);
3935                                         }
3936                                 }
3937                                 (void) strlcat(ringstr, ringsubstr,
3938                                     RINGSTRLEN);
3939                                 start = index;
3940                                 end = index;
3941                         }
3942                 }
3943                 /* The last one */
3944                 if (start != -1) {
3945                         if (first) {
3946                                 if (start == end) {
3947                                         (void) snprintf(buf, bufsize, "%d",
3948                                             start);
3949                                 } else {
3950                                         (void) snprintf(buf, bufsize, "%d-%d",
3951                                             start, end);
3952                                 }
3953                         } else {
3954                                 if (start == end) {
3955                                         (void) snprintf(ringsubstr, RINGSTRLEN,
3956                                             ",%d", start);
3957                                 } else {
3958                                         (void) snprintf(ringsubstr, RINGSTRLEN,
3959                                             ",%d-%d", start, end);
3960                                 }
3961                                 (void) strlcat(ringstr, ringsubstr, RINGSTRLEN);
3962                                 (void) snprintf(buf, bufsize, "%s", ringstr);
3963                         }
3964                 }
3965                 break;
3966         case PHYS_H_CLIENTS:
```

```
3967                 if (attr->hg_client_names[0] == '\0') {
3968                         (void) snprintf(buf, bufsize, "--");
3969                 } else {
3970                         (void) snprintf(buf, bufsize, "%s ",
3971                             attr->hg_client_names);
3972                 }
3973                 break;
3974         }

3976         return (B_TRUE);
3977 }

3979 /*
3980  * callback for dladm_walk_macaddr, invoked for each MAC address slot
3981  */
3982 static boolean_t
3983 print_phys_mac_callback(void *arg, dladm_macaddr_attr_t *attr)
3984 {
3985         print_phys_mac_state_t *mac_state = arg;
3986         show_state_t *state = mac_state->ms_state;

3988         mac_state->ms_mac_attr = attr;
3989         ofmt_print(state->ls_ofmt, mac_state);

3991         return (B_TRUE);
3992 }

3994 /*
3995  * invoked by show-phys -m for each physical data-link
3996  */
3997 static dladm_status_t
3998 print_phys_mac(show_state_t *state, datalink_id_t linkid, char *link)
3999 {
4000         print_phys_mac_state_t mac_state;

4002         mac_state.ms_state = state;
4003         mac_state.ms_link = link;

4005         return (dladm_walk_macaddr(handle, linkid, &mac_state,
4006             print_phys_mac_callback));
4007 }

4009 /*
4010  * callback for dladm_walk_hwgrp, invoked for each MAC hwgrp
4011  */
4012 static boolean_t
4013 print_phys_hwgrp_callback(void *arg, dladm_hwgrp_attr_t *attr)
4014 {
4015         print_phys_hwgrp_state_t *hwgrp_state = arg;
4016         show_state_t *state = hwgrp_state->hs_state;

4018         hwgrp_state->hs_grp_attr = attr;
4019         ofmt_print(state->ls_ofmt, hwgrp_state);

4021         return (B_TRUE);
4022 }

4024 /* invoked by show-phys -H for each physical data-link */
4025 static dladm_status_t
4026 print_phys_hwgrp(show_state_t *state, datalink_id_t linkid, char *link)
4027 {
4028         print_phys_hwgrp_state_t hwgrp_state;

4030         hwgrp_state.hs_state = state;
4031         hwgrp_state.hs_link = link;
4032         return (dladm_walk_hwgrp(handle, linkid, &hwgrp_state,
```



```
4033             print_phys_hwgrp_callback));
4034 }

4036 /*
4037  * Parse the "local=<laddr>,remote=<raddr>" sub-options for the -a option of
4038  * *-iptun subcommands.
4039  */
4040 static void
4041 iptun_process_addrarg(char *addrarg, iptun_params_t *params)
4042 {
4043         char *addrval;

4045         while (*addrarg != '\0') {
4046                 switch (getsubopt(&addrarg, iptun_addropts, &addrval)) {
4047                 case IPTUN_LOCAL:
4048                         params->iptun_param_flags |= IPTUN_PARAM_LADDR;
4049                         if (strlcpy(params->iptun_param_laddr, addrval,
4050                             sizeof (params->iptun_param_laddr)) >=
4051                             sizeof (params->iptun_param_laddr))
4052                                 die("tunnel source address is too long");
4053                         break;
4054                 case IPTUN_REMOTE:
4055                         params->iptun_param_flags |= IPTUN_PARAM_RADDR;
4056                         if (strlcpy(params->iptun_param_raddr, addrval,
4057                             sizeof (params->iptun_param_raddr)) >=
4058                             sizeof (params->iptun_param_raddr))
4059                                 die("tunnel destination address is too long");
4060                         break;
4061                 default:
4062                         die("invalid address type: %s", addrval);
4063                         break;
4064                 }
4065         }
4066 }

4068 /*
4069  * Convenience routine to process iptun-create/modify/delete subcommand
4070  * arguments.
4071  */
4072 static void
4073 iptun_process_args(int argc, char *argv[], const char *opts,
4074     iptun_params_t *params, uint32_t *flags, char *name, const char *use)
4075 {
4076         int     option;
4077         char    *altroot = NULL;

4079         if (params != NULL)
4080                 bzero(params, sizeof (*params));
4081         *flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;

4083         opterr = 0;
4084         while ((option = getopt_long(argc, argv, opts, iptun_lopts, NULL)) !=
4085             -1) {
4086                 switch (option) {
4087                 case 'a':
4088                         iptun_process_addrarg(optarg, params);
4089                         break;
4090                 case 'R':
4091                         altroot = optarg;
4092                         break;
4093                 case 't':
4094                         *flags &= ~DLADM_OPT_PERSIST;
4095                         break;
4096                 case 'T':
4097                         params->iptun_param_type = iptun_gettypebyname(optarg);
4098                         if (params->iptun_param_type == IPTUN_TYPE_UNKNOWN)
```

```
4099                                die("unknown tunnel type: %s", optarg);
4100                        params->iptun_param_flags |= IPTUN_PARAM_TYPE;
4101                        break;
4102                default:
4103                        die_opterr(optopt, option, use);
4104                        break;
4105                }
4106        }

4108        /* Get the required tunnel name argument. */
4109        if (argc - optind != 1)
4110                usage();

4112        if (strlcpy(name, argv[optind], MAXLINKNAMELEN) >= MAXLINKNAMELEN)
4113                die("tunnel name is too long");

4115        if (altroot != NULL)
4116                altroot_cmd(altroot, argc, argv);
4117 }

4119 static void
4120 do_create_iptun(int argc, char *argv[], const char *use)
4121 {
4122        iptun_params_t  params;
4123        dladm_status_t  status;
4124        uint32_t        flags;
4125        char            name[MAXLINKNAMELEN];

4127        iptun_process_args(argc, argv, ":a:R:tT:", &params, &flags, name,
4128            use);

4130        status = dladm_iptun_create(handle, name, &params, flags);
4131        if (status != DLADM_STATUS_OK)
4132                die_dlerr(status, "could not create tunnel");
4133 }

4135 static void
4136 do_delete_iptun(int argc, char *argv[], const char *use)
4137 {
4138        uint32_t        flags;
4139        datalink_id_t   linkid;
4140        dladm_status_t  status;
4141        char            name[MAXLINKNAMELEN];

4143        iptun_process_args(argc, argv, ":R:t", NULL, &flags, name, use);

4145        status = dladm_name2info(handle, name, &linkid, NULL, NULL, NULL);
4146        if (status != DLADM_STATUS_OK)
4147                die_dlerr(status, "could not delete tunnel");
4148        status = dladm_iptun_delete(handle, linkid, flags);
4149        if (status != DLADM_STATUS_OK)
4150                die_dlerr(status, "could not delete tunnel");
4151 }

4153 static void
4154 do_modify_iptun(int argc, char *argv[], const char *use)
4155 {
4156        iptun_params_t  params;
4157        uint32_t        flags;
4158        dladm_status_t  status;
4159        char            name[MAXLINKNAMELEN];

4161        iptun_process_args(argc, argv, ":a:R:t", &params, &flags, name, use);

4163        if ((status = dladm_name2info(handle, name, &params.iptun_param_linkid,
4164            NULL, NULL, NULL)) != DLADM_STATUS_OK)
```



```
4165                die_dlerr(status, "could not modify tunnel");
4166        status = dladm_iptun_modify(handle, &params, flags);
4167        if (status != DLADM_STATUS_OK)
4168                die_dlerr(status, "could not modify tunnel");
4169 }

4171 static void
4172 do_show_iptun(int argc, char *argv[], const char *use)
4173 {
4174        char            option;
4175        datalink_id_t   linkid;
4176        uint32_t        flags = DLADM_OPT_ACTIVE;
4177        char            *name = NULL;
4178        dladm_status_t  status;
4179        const char      *fields_str = NULL;
4180        show_state_t    state;
4181        ofmt_handle_t   ofmt;
4182        ofmt_status_t   oferr;
4183        uint_t          ofmtflags = 0;

4185        bzero(&state, sizeof (state));
4186        opterr = 0;
4187        while ((option = getopt_long(argc, argv, ":pPo:",
4188            iptun_lopts, NULL)) != -1) {
4189                switch (option) {
4190                case 'o':
4191                        fields_str = optarg;
4192                        break;
4193                case 'p':
4194                        state.ls_parsable = B_TRUE;
4195                        ofmtflags = OFMT_PARSABLE;
4196                        break;
4197                case 'P':
4198                        flags = DLADM_OPT_PERSIST;
4199                        break;
4200                default:
4201                        die_opterr(optopt, option, use);
4202                        break;
4203                }
4204        }

4206        /*
4207         * Get the optional tunnel name argument.  If there is one, it must
4208         * be the last thing remaining on the command-line.
4209         */
4210        if (argc - optind > 1)
4211                die(gettext(use));
4212        if (argc - optind == 1)
4213                name = argv[optind];

4215        oferr = ofmt_open(fields_str, iptun_fields, ofmtflags,
4216            DLADM_DEFAULT_COL, &ofmt);
4217        dladm_ofmt_check(oferr, state.ls_parsable, ofmt);

4219        state.ls_ofmt = ofmt;
4220        state.ls_flags = flags;

4222        if (name == NULL) {
4223                (void) dladm_walk_datalink_id(print_iptun_walker, handle,
4224                    &state, DATALINK_CLASS_IPTUN, DATALINK_ANY_MEDIATYPE,
4225                    flags);
4226                status = state.ls_status;
4227        } else {
4228                if ((status = dladm_name2info(handle, name, &linkid, NULL, NULL,
4229                    NULL)) == DLADM_STATUS_OK)
4230                        status = print_iptun(handle, linkid, &state);
```

```
4231         }

4233         if (status != DLADM_STATUS_OK)
4234                 die_dlerr(status, "unable to obtain tunnel status");
4235 }

4237 /* ARGSUSED */
4238 static void
4239 do_up_iptun(int argc, char *argv[], const char *use)
4240 {
4241         datalink_id_t   linkid = DATALINK_ALL_LINKID;
4242         dladm_status_t  status = DLADM_STATUS_OK;

4244         /*
4245          * Get the optional tunnel name argument.  If there is one, it must
4246          * be the last thing remaining on the command-line.
4247          */
4248         if (argc - optind > 1)
4249                 usage();
4250         if (argc - optind == 1) {
4251                 status = dladm_name2info(handle, argv[optind], &linkid, NULL,
4252                     NULL, NULL);
4253         }
4254         if (status == DLADM_STATUS_OK)
4255                 status = dladm_iptun_up(handle, linkid);
4256         if (status != DLADM_STATUS_OK)
4257                 die_dlerr(status, "unable to configure IP tunnel links");
4258 }

4260 /* ARGSUSED */
4261 static void
4262 do_down_iptun(int argc, char *argv[], const char *use)
4263 {
4264         datalink_id_t   linkid = DATALINK_ALL_LINKID;
4265         dladm_status_t  status = DLADM_STATUS_OK;

4267         /*
4268          * Get the optional tunnel name argument.  If there is one, it must
4269          * be the last thing remaining on the command-line.
4270          */
4271         if (argc - optind > 1)
4272                 usage();
4273         if (argc - optind == 1) {
4274                 status = dladm_name2info(handle, argv[optind], &linkid, NULL,
4275                     NULL, NULL);
4276         }
4277         if (status == DLADM_STATUS_OK)
4278                 status = dladm_iptun_down(handle, linkid);
4279         if (status != DLADM_STATUS_OK)
4280                 die_dlerr(status, "unable to bring down IP tunnel links");
4281 }

4283 static iptun_type_t
4284 iptun_gettypebyname(char *typestr)
4285 {
4286         int i;

4288         for (i = 0; iptun_types[i].type_name != NULL; i++) {
4289                 if (strncmp(iptun_types[i].type_name, typestr,
4290                     strlen(iptun_types[i].type_name)) == 0) {
4291                         return (iptun_types[i].type_value);
4292                 }
4293         }
4294         return (IPTUN_TYPE_UNKNOWN);
4295 }
```



```
4297 static const char *
4298 iptun_gettypebyvalue(iptun_type_t type)
4299 {
4300         int i;

4302         for (i = 0; iptun_types[i].type_name != NULL; i++) {
4303                 if (iptun_types[i].type_value == type)
4304                         return (iptun_types[i].type_name);
4305         }
4306         return (NULL);
4307 }

4309 static dladm_status_t
4310 print_iptun(dladm_handle_t dh, datalink_id_t linkid, show_state_t *state)
4311 {
4312         dladm_status_t          status;
4313         iptun_params_t          params;
4314         iptun_fields_buf_t      lbuf;
4315         const char              *laddr;
4316         const char              *raddr;

4318         params.iptun_param_linkid = linkid;
4319         status = dladm_iptun_getparams(dh, &params, state->ls_flags);
4320         if (status != DLADM_STATUS_OK)
4321                 return (status);

4323         /* LINK */
4324         status = dladm_datalink_id2info(dh, linkid, NULL, NULL, NULL,
4325             lbuf.iptun_name, sizeof (lbuf.iptun_name));
4326         if (status != DLADM_STATUS_OK)
4327                 return (status);

4329         /* TYPE */
4330         (void) strlcpy(lbuf.iptun_type,
4331             iptun_gettypebyvalue(params.iptun_param_type),
4332             sizeof (lbuf.iptun_type));

4334         /* FLAGS */
4335         (void) memset(lbuf.iptun_flags, '-', IPTUN_NUM_FLAGS);
4336         lbuf.iptun_flags[IPTUN_NUM_FLAGS] = '\0';
4337         if (params.iptun_param_flags & IPTUN_PARAM_IPSECPOL)
4338                 lbuf.iptun_flags[IPTUN_SFLAG_INDEX] = 's';
4339         if (params.iptun_param_flags & IPTUN_PARAM_IMPLICIT)
4340                 lbuf.iptun_flags[IPTUN_IFLAG_INDEX] = 'i';

4342         /* LOCAL */
4343         if (params.iptun_param_flags & IPTUN_PARAM_LADDR)
4344                 laddr = params.iptun_param_laddr;
4345         else
4346                 laddr = (state->ls_parsable) ? "" : "--";
4347         (void) strlcpy(lbuf.iptun_laddr, laddr, sizeof (lbuf.iptun_laddr));

4349         /* REMOTE */
4350         if (params.iptun_param_flags & IPTUN_PARAM_RADDR)
4351                 raddr = params.iptun_param_raddr;
4352         else
4353                 raddr = (state->ls_parsable) ? "" : "--";
4354         (void) strlcpy(lbuf.iptun_raddr, raddr, sizeof (lbuf.iptun_raddr));

4356         ofmt_print(state->ls_ofmt, &lbuf);

4358         return (DLADM_STATUS_OK);
4359 }

4361 static int
4362 print_iptun_walker(dladm_handle_t dh, datalink_id_t linkid, void *arg)
```

```
4363 {
4364         ((show_state_t *)arg)->ls_status = print_iptun(dh, linkid, arg);
4365         return (DLADM_WALK_CONTINUE);
4366 }

4368 static dladm_status_t
4369 print_phys(show_state_t *state, datalink_id_t linkid)
4370 {
4371         char                    link[MAXLINKNAMELEN];
4372         uint32_t                flags;
4373         dladm_status_t          status;
4374         datalink_class_t        class;
4375         uint32_t                media;

4377         if ((status = dladm_datalink_id2info(handle, linkid, &flags, &class,
4378             &media, link, MAXLINKNAMELEN)) != DLADM_STATUS_OK) {
4379                 goto done;
4380         }

4382         if (class != DATALINK_CLASS_PHYS) {
4383                 status = DLADM_STATUS_BADARG;
4384                 goto done;
4385         }

4387         if (!(state->ls_flags & flags)) {
4388                 status = DLADM_STATUS_NOTFOUND;
4389                 goto done;
4390         }

4392         if (state->ls_mac)
4393                 status = print_phys_mac(state, linkid, link);
4394         else if (state->ls_hwgrp)
4395                 status = print_phys_hwgrp(state, linkid, link);
4396         else
4397                 status = print_phys_default(state, linkid, link, flags, media);

4399 done:
4400         return (status);
4401 }

4403 /* ARGSUSED */
4404 static int
4405 show_phys(dladm_handle_t dh, datalink_id_t linkid, void *arg)
4406 {
4407         show_state_t    *state = arg;

4409         state->ls_status = print_phys(state, linkid);
4410         return (DLADM_WALK_CONTINUE);
4411 }

4413 /*
4414  * Print the active topology information.
4415  */
4416 static dladm_status_t
4417 print_vlan(show_state_t *state, datalink_id_t linkid, link_fields_buf_t *l)
4418 {
4419         dladm_vlan_attr_t       vinfo;
4420         uint32_t                flags;
4421         dladm_status_t          status;

4423         if ((status = dladm_datalink_id2info(handle, linkid, &flags, NULL, NULL,
4424             l->link_name, sizeof (l->link_name))) != DLADM_STATUS_OK) {
4425                 goto done;
4426         }

4428         if (!(state->ls_flags & flags)) {
```



```
4429                 status = DLADM_STATUS_NOTFOUND;
4430                 goto done;
4431         }

4433         if ((status = dladm_vlan_info(handle, linkid, &vinfo,
4434             state->ls_flags)) != DLADM_STATUS_OK ||
4435             (status = dladm_datalink_id2info(handle, vinfo.dv_linkid, NULL,
4436             NULL, NULL, l->link_over, sizeof (l->link_over))) !=
4437             DLADM_STATUS_OK) {
4438                 goto done;
4439         }

4441         (void) snprintf(l->link_vlan_vid, sizeof (l->link_vlan_vid), "%d",
4442             vinfo.dv_vid);
4443         (void) snprintf(l->link_flags, sizeof (l->link_flags), "%c----",
4444             vinfo.dv_force ? 'f' : '-');

4446 done:
4447         return (status);
4448 }

4450 /* ARGSUSED */
4451 static int
4452 show_vlan(dladm_handle_t dh, datalink_id_t linkid, void *arg)
4453 {
4454         show_state_t            *state = arg;
4455         dladm_status_t          status;
4456         link_fields_buf_t       lbuf;

4458         bzero(&lbuf, sizeof (link_fields_buf_t));
4459         status = print_vlan(state, linkid, &lbuf);
4460         if (status != DLADM_STATUS_OK)
4461                 goto done;

4463         ofmt_print(state->ls_ofmt, &lbuf);

4465 done:
4466         state->ls_status = status;
4467         return (DLADM_WALK_CONTINUE);
4468 }

4470 static void
4471 do_show_phys(int argc, char *argv[], const char *use)
4472 {
4473         int             option;
4474         uint32_t        flags = DLADM_OPT_ACTIVE;
4475         boolean_t       p_arg = B_FALSE;
4476         boolean_t       o_arg = B_FALSE;
4477         boolean_t       m_arg = B_FALSE;
4478         boolean_t       H_arg = B_FALSE;
4479         datalink_id_t   linkid = DATALINK_ALL_LINKID;
4480         show_state_t    state;
4481         dladm_status_t  status;
4482         char            *fields_str = NULL;
4483         char            *all_active_fields =
4484             "link,media,state,speed,duplex,device";
4485         char            *all_inactive_fields = "link,device,media,flags";
4486         char            *all_mac_fields = "link,slot,address,inuse,client";
4487         char            *all_hwgrp_fields = "link,ringtype,rings,clients";
4488         const ofmt_field_t *pf;
4489         ofmt_handle_t   ofmt;
4490         ofmt_status_t   oferr;
4491         uint_t          ofmtflags = 0;

4493         bzero(&state, sizeof (state));
4494         opterr = 0;
```

```
4495         while ((option = getopt_long(argc, argv, ":pPo:mH",
4496             show_lopts, NULL)) != -1) {
4497                 switch (option) {
4498                 case 'p':
4499                         if (p_arg)
4500                                 die_optdup(option);

4502                         p_arg = B_TRUE;
4503                         break;
4504                 case 'P':
4505                         if (flags != DLADM_OPT_ACTIVE)
4506                                 die_optdup(option);

4508                         flags = DLADM_OPT_PERSIST;
4509                         break;
4510                 case 'o':
4511                         o_arg = B_TRUE;
4512                         fields_str = optarg;
4513                         break;
4514                 case 'm':
4515                         m_arg = B_TRUE;
4516                         break;
4517                 case 'H':
4518                         H_arg = B_TRUE;
4519                         break;
4520                 default:
4521                         die_opterr(optopt, option, use);
4522                         break;
4523                 }
4524         }

4526         if (p_arg && !o_arg)
4527                 die("-p requires -o");

4529         if (m_arg && H_arg)
4530                 die("-m cannot combine with -H");

4532         if (p_arg && strcasecmp(fields_str, "all") == 0)
4533                 die("\"-o all\" is invalid with -p");

4535         /* get link name (optional last argument) */
4536         if (optind == (argc-1)) {
4537                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
4538                     NULL, NULL, NULL)) != DLADM_STATUS_OK) {
4539                         die_dlerr(status, "link %s is not valid", argv[optind]);
4540                 }
4541         } else if (optind != argc) {
4542                 usage();
4543         }

4545         state.ls_parsable = p_arg;
4546         state.ls_flags = flags;
4547         state.ls_donefirst = B_FALSE;
4548         state.ls_mac = m_arg;
4549         state.ls_hwgrp = H_arg;

4551         if (m_arg && !(flags & DLADM_OPT_ACTIVE)) {
4552                 /*
4553                  * We can only display the factory MAC addresses of
4554                  * active data-links.
4555                  */
4556                 die("-m not compatible with -P");
4557         }

4559         if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0)) {
4560                 if (state.ls_mac)
```



```
4561                         fields_str = all_mac_fields;
4562                 else if (state.ls_hwgrp)
4563                         fields_str = all_hwgrp_fields;
4564                 else if (state.ls_flags & DLADM_OPT_ACTIVE) {
4565                         fields_str = all_active_fields;
4566                 } else {
4567                         fields_str = all_inactive_fields;
4568                 }
4569         }

4571         if (state.ls_mac) {
4572                 pf = phys_m_fields;
4573         } else if (state.ls_hwgrp) {
4574                 pf = phys_h_fields;
4575         } else {
4576                 pf = phys_fields;
4577         }

4579         if (state.ls_parsable)
4580                 ofmtflags |= OFMT_PARSABLE;
4581         oferr = ofmt_open(fields_str, pf, ofmtflags, 0, &ofmt);
4582         dladm_ofmt_check(oferr, state.ls_parsable, ofmt);
4583         state.ls_ofmt = ofmt;

4585         if (linkid == DATALINK_ALL_LINKID) {
4586                 (void) dladm_walk_datalink_id(show_phys, handle, &state,
4587                     DATALINK_CLASS_PHYS, DATALINK_ANY_MEDIATYPE, flags);
4588         } else {
4589                 (void) show_phys(handle, linkid, &state);
4590                 if (state.ls_status != DLADM_STATUS_OK) {
4591                         die_dlerr(state.ls_status,
4592                             "failed to show physical link %s", argv[optind]);
4593                 }
4594         }
4595         ofmt_close(ofmt);
4596 }

4598 static void
4599 do_show_vlan(int argc, char *argv[], const char *use)
4600 {
4601         int             option;
4602         uint32_t        flags = DLADM_OPT_ACTIVE;
4603         boolean_t       p_arg = B_FALSE;
4604         datalink_id_t   linkid = DATALINK_ALL_LINKID;
4605         show_state_t    state;
4606         dladm_status_t  status;
4607         boolean_t       o_arg = B_FALSE;
4608         char            *fields_str = NULL;
4609         ofmt_handle_t   ofmt;
4610         ofmt_status_t   oferr;
4611         uint_t          ofmtflags = 0;

4613         bzero(&state, sizeof (state));

4615         opterr = 0;
4616         while ((option = getopt_long(argc, argv, ":pPo:",
4617             show_lopts, NULL)) != -1) {
4618                 switch (option) {
4619                 case 'p':
4620                         if (p_arg)
4621                                 die_optdup(option);

4623                         p_arg = B_TRUE;
4624                         break;
4625                 case 'P':
4626                         if (flags != DLADM_OPT_ACTIVE)
```

```
4627                                    die_optdup(option);

4629                            flags = DLADM_OPT_PERSIST;
4630                            break;
4631                    case 'o':
4632                            o_arg = B_TRUE;
4633                            fields_str = optarg;
4634                            break;
4635                    default:
4636                            die_opterr(optopt, option, use);
4637                            break;
4638                    }
4639            }

4641            /* get link name (optional last argument) */
4642            if (optind == (argc-1)) {
4643                    if ((status = dladm_name2info(handle, argv[optind], &linkid,
4644                        NULL, NULL, NULL)) != DLADM_STATUS_OK) {
4645                            die_dlerr(status, "link %s is not valid", argv[optind]);
4646                    }
4647            } else if (optind != argc) {
4648                    usage();
4649            }

4651            state.ls_parsable = p_arg;
4652            state.ls_flags = flags;
4653            state.ls_donefirst = B_FALSE;

4655            if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0))
4656                    fields_str = NULL;

4658            if (state.ls_parsable)
4659                    ofmtflags |= OFMT_PARSABLE;
4660            oferr = ofmt_open(fields_str, vlan_fields, ofmtflags, 0, &ofmt);
4661            dladm_ofmt_check(oferr, state.ls_parsable, ofmt);
4662            state.ls_ofmt = ofmt;

4664            if (linkid == DATALINK_ALL_LINKID) {
4665                    (void) dladm_walk_datalink_id(show_vlan, handle, &state,
4666                        DATALINK_CLASS_VLAN, DATALINK_ANY_MEDIATYPE, flags);
4667            } else {
4668                    (void) show_vlan(handle, linkid, &state);
4669                    if (state.ls_status != DLADM_STATUS_OK) {
4670                            die_dlerr(state.ls_status, "failed to show vlan %s",
4671                                argv[optind]);
4672                    }
4673            }
4674            ofmt_close(ofmt);
4675    }

4677    static void
4678    do_create_vnic(int argc, char *argv[], const char *use)
4679    {
4680            datalink_id_t           linkid, dev_linkid;
4681            char                    devname[MAXLINKNAMELEN];
4682            char                    name[MAXLINKNAMELEN];
4683            boolean_t               l_arg = B_FALSE;
4684            uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
4685            char                    *altroot = NULL;
4686            int                     option;
4687            char                    *endp = NULL;
4688            dladm_status_t          status;
4689            vnic_mac_addr_type_t    mac_addr_type = VNIC_MAC_ADDR_TYPE_UNKNOWN;
4690            uchar_t                 *mac_addr = NULL;
4691            int                     mac_slot = -1;
4692            uint_t                  maclen = 0, mac_prefix_len = 0;
```



```
4693            char                    propstr[DLADM_STRSIZE];
4694            dladm_arg_list_t        *proplist = NULL;
4695            int                     vid = 0;
4696            int                     af = AF_UNSPEC;
4697            vrid_t                  vrid = VRRP_VRID_NONE;

4699            opterr = 0;
4700            bzero(propstr, DLADM_STRSIZE);

4702            while ((option = getopt_long(argc, argv, ":tfR:l:m:n:p:r:v:V:A:H",
4703                vnic_lopts, NULL)) != -1) {
4704                    switch (option) {
4705                    case 't':
4706                            flags &= ~DLADM_OPT_PERSIST;
4707                            break;
4708                    case 'R':
4709                            altroot = optarg;
4710                            break;
4711                    case 'l':
4712                            if (strlcpy(devname, optarg, MAXLINKNAMELEN) >=
4713                                MAXLINKNAMELEN)
4714                                    die("link name too long");
4715                            l_arg = B_TRUE;
4716                            break;
4717                    case 'm':
4718                            if (mac_addr_type != VNIC_MAC_ADDR_TYPE_UNKNOWN)
4719                                    die("cannot specify -m option twice");

4721                            if (strcmp(optarg, "fixed") == 0) {
4722                                    /*
4723                                     * A fixed MAC address must be specified
4724                                     * by its value, not by the keyword 'fixed'.
4725                                     */
4726                                    die("'fixed' is not a valid MAC address");
4727                            }
4728                            if (dladm_vnic_str2macaddrtype(optarg,
4729                                &mac_addr_type) != DLADM_STATUS_OK) {
4730                                    mac_addr_type = VNIC_MAC_ADDR_TYPE_FIXED;
4731                                    /* MAC address specified by value */
4732                                    mac_addr = _link_aton(optarg, (int *)&maclen);
4733                                    if (mac_addr == NULL) {
4734                                            if (maclen == (uint_t)-1)
4735                                                    die("invalid MAC address");
4736                                            else
4737                                                    die("out of memory");
4738                                    }
4739                            }
4740                            break;
4741                    case 'n':
4742                            errno = 0;
4743                            mac_slot = (int)strtol(optarg, &endp, 10);
4744                            if (errno != 0 || *endp != '\0')
4745                                    die("invalid slot number");
4746                            break;
4747                    case 'p':
4748                            (void) strlcat(propstr, optarg, DLADM_STRSIZE);
4749                            if (strlcat(propstr, ",", DLADM_STRSIZE) >=
4750                                DLADM_STRSIZE)
4751                                    die("property list too long '%s'", propstr);
4752                            break;
4753                    case 'r':
4754                            mac_addr = _link_aton(optarg, (int *)&mac_prefix_len);
4755                            if (mac_addr == NULL) {
4756                                    if (mac_prefix_len == (uint_t)-1)
4757                                            die("invalid MAC address");
4758                                    else
```

```
4759                                        die("out of memory");
4760                        }
4761                        break;
4762                case 'V':
4763                        if (!str2int(optarg, (int *)&vrid) ||
4764                            vrid < VRRP_VRID_MIN || vrid > VRRP_VRID_MAX) {
4765                                die("invalid VRRP identifier '%s'", optarg);
4766                        }
4767
4768                        break;
4769                case 'A':
4770                        if (strcmp(optarg, "inet") == 0)
4771                                af = AF_INET;
4772                        else if (strcmp(optarg, "inet6") == 0)
4773                                af = AF_INET6;
4774                        else
4775                                die("invalid address family '%s'", optarg);
4776                        break;
4777                case 'v':
4778                        if (vid != 0)
4779                                die_optdup(option);
4780
4781                        if (!str2int(optarg, &vid) || vid < 1 || vid > 4094)
4782                                die("invalid VLAN identifier '%s'", optarg);
4783
4784                        break;
4785                case 'f':
4786                        flags |= DLADM_OPT_FORCE;
4787                        break;
4788                default:
4789                        die_opterr(optopt, option, use);
4790                }
4791        }
4792
4793        if (mac_addr_type == VNIC_MAC_ADDR_TYPE_UNKNOWN)
4794                mac_addr_type = VNIC_MAC_ADDR_TYPE_AUTO;
4795
4796        /*
4797         * 'f' - force, flag can be specified only with 'v' - vlan.
4798         */
4799        if ((flags & DLADM_OPT_FORCE) != 0 && vid == 0)
4800                die("-f option can only be used with -v");
4801
4802        if (mac_prefix_len != 0 && mac_addr_type != VNIC_MAC_ADDR_TYPE_RANDOM &&
4803            mac_addr_type != VNIC_MAC_ADDR_TYPE_FIXED)
4804                usage();
4805
4806        if (mac_addr_type == VNIC_MAC_ADDR_TYPE_VRID) {
4807                if (vrid == VRRP_VRID_NONE || af == AF_UNSPEC ||
4808                    mac_addr != NULL || maclen != 0 || mac_slot != -1 ||
4809                    mac_prefix_len != 0) {
4810                        usage();
4811                }
4812        } else if ((af != AF_UNSPEC || vrid != VRRP_VRID_NONE)) {
4813                usage();
4814        }
4815
4816        /* check required options */
4817        if (!l_arg)
4818                usage();
4819
4820        if (mac_slot != -1 && mac_addr_type != VNIC_MAC_ADDR_TYPE_FACTORY)
4821                usage();
4822
4823        /* the VNIC id is the required operand */
4824        if (optind != (argc - 1))
```



```
4825                usage();
4826
4827        if (strlcpy(name, argv[optind], MAXLINKNAMELEN) >= MAXLINKNAMELEN)
4828                die("link name too long '%s'", argv[optind]);
4829
4830        if (!dladm_valid_linkname(name))
4831                die("invalid link name '%s'", argv[optind]);
4832
4833        if (altroot != NULL)
4834                altroot_cmd(altroot, argc, argv);
4835
4836        if (dladm_name2info(handle, devname, &dev_linkid, NULL, NULL, NULL) !=
4837            DLADM_STATUS_OK)
4838                die("invalid link name '%s'", devname);
4839
4840        if (dladm_parse_link_props(propstr, &proplist, B_FALSE)
4841            != DLADM_STATUS_OK)
4842                die("invalid vnic property");
4843
4844        status = dladm_vnic_create(handle, name, dev_linkid, mac_addr_type,
4845            mac_addr, maclen, &mac_slot, mac_prefix_len, vid, vrid, af,
4846            &linkid, proplist, flags);
4847        switch (status) {
4848        case DLADM_STATUS_OK:
4849                break;
4850
4851        case DLADM_STATUS_LINKBUSY:
4852                die("VLAN over '%s' may not use default_tag ID "
4853                    "(see dladm(1M))", devname);
4854                break;
4855
4856        default:
4857                die_dlerr(status, "vnic creation over %s failed", devname);
4858        }
4859
4860        dladm_free_props(proplist);
4861        free(mac_addr);
4862 }
4863
4864 static void
4865 do_etherstub_check(const char *name, datalink_id_t linkid, boolean_t etherstub,
4866     uint32_t flags)
4867 {
4868        boolean_t is_etherstub;
4869        dladm_vnic_attr_t attr;
4870
4871        if (dladm_vnic_info(handle, linkid, &attr, flags) != DLADM_STATUS_OK) {
4872                /*
4873                 * Let the delete continue anyway.
4874                 */
4875                return;
4876        }
4877        is_etherstub = (attr.va_link_id == DATALINK_INVALID_LINKID);
4878        if (is_etherstub != etherstub) {
4879                die("'%s' is not %s", name,
4880                    (is_etherstub ? "a vnic" : "an etherstub"));
4881        }
4882 }
4883
4884 static void
4885 do_delete_vnic_common(int argc, char *argv[], const char *use,
4886     boolean_t etherstub)
4887 {
4888        int option;
4889        uint32_t flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
4890        datalink_id_t linkid;
```

```
4891         char *altroot = NULL;
4892         dladm_status_t status;

4894         opterr = 0;
4895         while ((option = getopt_long(argc, argv, ":R:t", lopts,
4896             NULL)) != -1) {
4897                 switch (option) {
4898                 case 't':
4899                         flags &= ~DLADM_OPT_PERSIST;
4900                         break;
4901                 case 'R':
4902                         altroot = optarg;
4903                         break;
4904                 default:
4905                         die_opterr(optopt, option, use);
4906                 }
4907         }

4909         /* get vnic name (required last argument) */
4910         if (optind != (argc - 1))
4911                 usage();

4913         if (altroot != NULL)
4914                 altroot_cmd(altroot, argc, argv);

4916         status = dladm_name2info(handle, argv[optind], &linkid, NULL, NULL,
4917             NULL);
4918         if (status != DLADM_STATUS_OK)
4919                 die("invalid link name '%s'", argv[optind]);

4921         if ((flags & DLADM_OPT_ACTIVE) != 0) {
4922                 do_etherstub_check(argv[optind], linkid, etherstub,
4923                     DLADM_OPT_ACTIVE);
4924         }
4925         if ((flags & DLADM_OPT_PERSIST) != 0) {
4926                 do_etherstub_check(argv[optind], linkid, etherstub,
4927                     DLADM_OPT_PERSIST);
4928         }

4930         status = dladm_vnic_delete(handle, linkid, flags);
4931         if (status != DLADM_STATUS_OK)
4932                 die_dlerr(status, "vnic deletion failed");
4933 }

4935 static void
4936 do_delete_vnic(int argc, char *argv[], const char *use)
4937 {
4938         do_delete_vnic_common(argc, argv, use, B_FALSE);
4939 }

4941 /* ARGSUSED */
4942 static void
4943 do_up_vnic_common(int argc, char *argv[], const char *use, boolean_t vlan)
4944 {
4945         datalink_id_t   linkid = DATALINK_ALL_LINKID;
4946         dladm_status_t  status;
4947         char            *type;

4949         type = vlan ? "vlan" : "vnic";

4951         /*
4952          * get the id or the name of the vnic/vlan (optional last argument)
4953          */
4954         if (argc == 2) {
4955                 status = dladm_name2info(handle, argv[1], &linkid, NULL, NULL,
4956                     NULL);
```



```
4957                 if (status != DLADM_STATUS_OK)
4958                         goto done;

4960         } else if (argc > 2) {
4961                 usage();
4962         }

4964         if (vlan)
4965                 status = dladm_vlan_up(handle, linkid);
4966         else
4967                 status = dladm_vnic_up(handle, linkid, 0);

4969 done:
4970         if (status != DLADM_STATUS_OK) {
4971                 if (argc == 2) {
4972                         die_dlerr(status,
4973                             "could not bring up %s '%s'", type, argv[1]);
4974                 } else {
4975                         die_dlerr(status, "could not bring %ss up", type);
4976                 }
4977         }
4978 }

4980 static void
4981 do_up_vnic(int argc, char *argv[], const char *use)
4982 {
4983         do_up_vnic_common(argc, argv, use, B_FALSE);
4984 }

4986 static void
4987 dump_vnics_head(const char *dev)
4988 {
4989         if (strlen(dev))
4990                 (void) printf("%s", dev);

4992         (void) printf("\tipackets  rbytes      opackets  obytes          ");

4994         if (strlen(dev))
4995                 (void) printf("%%ipkts  %%opkts\n");
4996         else
4997                 (void) printf("\n");
4998 }

5000 static void
5001 dump_vnic_stat(const char *name, datalink_id_t vnic_id,
5002     show_vnic_state_t *state, pktsum_t *vnic_stats, pktsum_t *tot_stats)
5003 {
5004         pktsum_t        diff_stats;
5005         pktsum_t        *old_stats = &state->vs_prevstats[vnic_id];

5007         dladm_stats_diff(&diff_stats, vnic_stats, old_stats);

5009         (void) printf("%s", name);

5011         (void) printf("\t%-10llu", diff_stats.ipackets);
5012         (void) printf("%-12llu", diff_stats.rbytes);
5013         (void) printf("%-10llu", diff_stats.opackets);
5014         (void) printf("%-12llu", diff_stats.obytes);

5016         if (tot_stats) {
5017                 if (tot_stats->ipackets == 0) {
5018                         (void) printf("\t-");
5019                 } else {
5020                         (void) printf("\t%-6.1f", (double)diff_stats.ipackets/
5021                             (double)tot_stats->ipackets * 100);
5022                 }
```

```
5023                 if (tot_stats->opackets == 0) {
5024                         (void) printf("\t-");
5025                 } else {
5026                         (void) printf("\t%-6.1f", (double)diff_stats.opackets/
5027                             (double)tot_stats->opackets * 100);
5028                 }
5029         }
5030         (void) printf("\n");

5032         *old_stats = *vnic_stats;
5033 }

5035 /*
5036  * Called from the walker dladm_vnic_walk_sys() for each vnic to display
5037  * vnic information or statistics.
5038  */
5039 static dladm_status_t
5040 print_vnic(show_vnic_state_t *state, datalink_id_t linkid)
5041 {
5042         dladm_vnic_attr_t       attr, *vnic = &attr;
5043         dladm_status_t          status;
5044         boolean_t               is_etherstub;
5045         char                    devname[MAXLINKNAMELEN];
5046         char                    vnic_name[MAXLINKNAMELEN];
5047         char                    mstr[MAXMACADDRLEN * 3];
5048         vnic_fields_buf_t       vbuf;

5050         if ((status = dladm_vnic_info(handle, linkid, vnic, state->vs_flags)) !=
5051             DLADM_STATUS_OK)
5052                 return (status);

5054         is_etherstub = (vnic->va_link_id == DATALINK_INVALID_LINKID);
5055         if (state->vs_etherstub != is_etherstub) {
5056                 /*
5057                  * Want all etherstub but it's not one, or want
5058                  * non-etherstub and it's one.
5059                  */
5060                 return (DLADM_STATUS_OK);
5061         }

5063         if (state->vs_link_id != DATALINK_ALL_LINKID) {
5064                 if (state->vs_link_id != vnic->va_link_id)
5065                         return (DLADM_STATUS_OK);
5066         }

5068         if (dladm_datalink_id2info(handle, linkid, NULL, NULL,
5069             NULL, vnic_name, sizeof (vnic_name)) != DLADM_STATUS_OK)
5070                 return (DLADM_STATUS_BADARG);

5072         bzero(devname, sizeof (devname));
5073         if (!is_etherstub &&
5074             dladm_datalink_id2info(handle, vnic->va_link_id, NULL, NULL,
5075             NULL, devname, sizeof (devname)) != DLADM_STATUS_OK)
5076                 (void) sprintf(devname, "?");

5078         state->vs_found = B_TRUE;
5079         if (state->vs_stats) {
5080                 /* print vnic statistics */
5081                 pktsum_t vnic_stats;

5083                 if (state->vs_firstonly) {
5084                         if (state->vs_donefirst)
5085                                 return (0);
5086                         state->vs_donefirst = B_TRUE;
5087                 }
```



```
5089                 if (!state->vs_printstats) {
5090                         /*
5091                          * get vnic statistics and add to the sum for the
5092                          * named device.
5093                          */
5094                         get_link_stats(vnic_name, &vnic_stats);
5095                         dladm_stats_total(&state->vs_totalstats, &vnic_stats,
5096                             &state->vs_prevstats[vnic->va_vnic_id]);
5097                 } else {
5098                         /* get and print vnic statistics */
5099                         get_link_stats(vnic_name, &vnic_stats);
5100                         dump_vnic_stat(vnic_name, linkid, state, &vnic_stats,
5101                             &state->vs_totalstats);
5102                 }
5103                 return (DLADM_STATUS_OK);
5104         } else {
5105                 (void) snprintf(vbuf.vnic_link, sizeof (vbuf.vnic_link),
5106                     "%s", vnic_name);

5108                 if (!is_etherstub) {

5110                         (void) snprintf(vbuf.vnic_over, sizeof (vbuf.vnic_over),
5111                             "%s", devname);
5112                         (void) snprintf(vbuf.vnic_speed,
5113                             sizeof (vbuf.vnic_speed), "%u",
5114                             (uint_t)((get_ifspeed(vnic_name, B_TRUE))
5115                             / 1000000ull));

5117                         switch (vnic->va_mac_addr_type) {
5118                         case VNIC_MAC_ADDR_TYPE_FIXED:
5119                         case VNIC_MAC_ADDR_TYPE_PRIMARY:
5120                                 (void) snprintf(vbuf.vnic_macaddrtype,
5121                                     sizeof (vbuf.vnic_macaddrtype),
5122                                     gettext("fixed"));
5123                                 break;
5124                         case VNIC_MAC_ADDR_TYPE_RANDOM:
5125                                 (void) snprintf(vbuf.vnic_macaddrtype,
5126                                     sizeof (vbuf.vnic_macaddrtype),
5127                                     gettext("random"));
5128                                 break;
5129                         case VNIC_MAC_ADDR_TYPE_FACTORY:
5130                                 (void) snprintf(vbuf.vnic_macaddrtype,
5131                                     sizeof (vbuf.vnic_macaddrtype),
5132                                     gettext("factory, slot %d"),
5133                                     vnic->va_mac_slot);
5134                                 break;
5135                         case VNIC_MAC_ADDR_TYPE_VRID:
5136                                 (void) snprintf(vbuf.vnic_macaddrtype,
5137                                     sizeof (vbuf.vnic_macaddrtype),
5138                                     gettext("vrrp, %d/%s"),
5139                                     vnic->va_vrid, vnic->va_af == AF_INET ?
5140                                     "inet" : "inet6");
5141                                 break;
5142                         }

5144                         if (strlen(vbuf.vnic_macaddrtype) > 0) {
5145                                 (void) snprintf(vbuf.vnic_macaddr,
5146                                     sizeof (vbuf.vnic_macaddr), "%s",
5147                                     dladm_aggr_macaddr2str(vnic->va_mac_addr,
5148                                     mstr));
5149                         }

5151                         (void) snprintf(vbuf.vnic_vid, sizeof (vbuf.vnic_vid),
5152                             "%d", vnic->va_vid);
5153                 }
```

```
5155                 ofmt_print(state->vs_ofmt, &vbuf);

5157                 return (DLADM_STATUS_OK);
5158         }
5159 }

5161 /* ARGSUSED */
5162 static int
5163 show_vnic(dladm_handle_t dh, datalink_id_t linkid, void *arg)
5164 {
5165         show_vnic_state_t       *state = arg;

5167         state->vs_status = print_vnic(state, linkid);
5168         return (DLADM_WALK_CONTINUE);
5169 }

5171 static void
5172 do_show_vnic_common(int argc, char *argv[], const char *use,
5173     boolean_t etherstub)
5174 {
5175         int                     option;
5176         boolean_t               s_arg = B_FALSE;
5177         boolean_t               i_arg = B_FALSE;
5178         boolean_t               l_arg = B_FALSE;
5179         uint32_t                interval = 0, flags = DLADM_OPT_ACTIVE;
5180         datalink_id_t           linkid = DATALINK_ALL_LINKID;
5181         datalink_id_t           dev_linkid = DATALINK_ALL_LINKID;
5182         show_vnic_state_t       state;
5183         dladm_status_t          status;
5184         boolean_t               o_arg = B_FALSE;
5185         char                    *fields_str = NULL;
5186         const ofmt_field_t      *pf;
5187         char                    *all_e_fields = "link";
5188         ofmt_handle_t           ofmt;
5189         ofmt_status_t           oferr;
5190         uint_t                  ofmtflags = 0;

5192         bzero(&state, sizeof (state));
5193         opterr = 0;
5194         while ((option = getopt_long(argc, argv, ":pPl:si:o:", lopts,
5195             NULL)) != -1) {
5196                 switch (option) {
5197                 case 'p':
5198                         state.vs_parsable = B_TRUE;
5199                         break;
5200                 case 'P':
5201                         flags = DLADM_OPT_PERSIST;
5202                         break;
5203                 case 'l':
5204                         if (etherstub)
5205                                 die("option not supported for this command");

5207                         if (strlcpy(state.vs_link, optarg, MAXLINKNAMELEN) >=
5208                             MAXLINKNAMELEN)
5209                                 die("link name too long");

5211                         l_arg = B_TRUE;
5212                         break;
5213                 case 's':
5214                         if (s_arg) {
5215                                 die("the option -s cannot be specified "
5216                                     "more than once");
5217                         }
5218                         s_arg = B_TRUE;
5219                         break;
5220                 case 'i':
```



```
5221                         if (i_arg) {
5222                                 die("the option -i cannot be specified "
5223                                     "more than once");
5224                         }
5225                         i_arg = B_TRUE;
5226                         if (!dladm_str2interval(optarg, &interval))
5227                                 die("invalid interval value '%s'", optarg);
5228                         break;
5229                 case 'o':
5230                         o_arg = B_TRUE;
5231                         fields_str = optarg;
5232                         break;
5233                 default:
5234                         die_opterr(optopt, option, use);
5235                 }
5236         }

5238         if (i_arg && !s_arg)
5239                 die("the option -i can be used only with -s");

5241         /* get vnic ID (optional last argument) */
5242         if (optind == (argc - 1)) {
5243                 status = dladm_name2info(handle, argv[optind], &linkid, NULL,
5244                     NULL, NULL);
5245                 if (status != DLADM_STATUS_OK) {
5246                         die_dlerr(status, "invalid vnic name '%s'",
5247                             argv[optind]);
5248                 }
5249                 (void) strlcpy(state.vs_vnic, argv[optind], MAXLINKNAMELEN);
5250         } else if (optind != argc) {
5251                 usage();
5252         }

5254         if (l_arg) {
5255                 status = dladm_name2info(handle, state.vs_link, &dev_linkid,
5256                     NULL, NULL, NULL);
5257                 if (status != DLADM_STATUS_OK) {
5258                         die_dlerr(status, "invalid link name '%s'",
5259                             state.vs_link);
5260                 }
5261         }

5263         state.vs_vnic_id = linkid;
5264         state.vs_link_id = dev_linkid;
5265         state.vs_etherstub = etherstub;
5266         state.vs_found = B_FALSE;
5267         state.vs_flags = flags;

5269         if (!o_arg || (o_arg && strcasecmp(fields_str, "all") == 0)) {
5270                 if (etherstub)
5271                         fields_str = all_e_fields;
5272         }
5273         pf = vnic_fields;

5275         if (state.vs_parsable)
5276                 ofmtflags |= OFMT_PARSABLE;
5277         oferr = ofmt_open(fields_str, pf, ofmtflags, 0, &ofmt);
5278         dladm_ofmt_check(oferr, state.vs_parsable, ofmt);
5279         state.vs_ofmt = ofmt;

5281         if (s_arg) {
5282                 /* Display vnic statistics */
5283                 vnic_stats(&state, interval);
5284                 ofmt_close(ofmt);
5285                 return;
5286         }
```

```
5288         /* Display vnic information */
5289         state.vs_donefirst = B_FALSE;

5291         if (linkid == DATALINK_ALL_LINKID) {
5292                 (void) dladm_walk_datalink_id(show_vnic, handle, &state,
5293                     DATALINK_CLASS_VNIC | DATALINK_CLASS_ETHERSTUB,
5294                     DATALINK_ANY_MEDIATYPE, flags);
5295         } else {
5296                 (void) show_vnic(handle, linkid, &state);
5297                 if (state.vs_status != DLADM_STATUS_OK) {
5298                         ofmt_close(ofmt);
5299                         die_dlerr(state.vs_status, "failed to show vnic '%s'",
5300                             state.vs_vnic);
5301                 }
5302         }
5303         ofmt_close(ofmt);
5304 }

5306 static void
5307 do_show_vnic(int argc, char *argv[], const char *use)
5308 {
5309         do_show_vnic_common(argc, argv, use, B_FALSE);
5310 }

5312 static void
5313 do_create_etherstub(int argc, char *argv[], const char *use)
5314 {
5315         uint32_t flags;
5316         char *altroot = NULL;
5317         int option;
5318         dladm_status_t status;
5319         char name[MAXLINKNAMELEN];
5320         uchar_t mac_addr[ETHERADDRL];

5322         name[0] = '\0';
5323         bzero(mac_addr, sizeof (mac_addr));
5324         flags = DLADM_OPT_ANCHOR | DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;

5326         opterr = 0;
5327         while ((option = getopt_long(argc, argv, "tR:",
5328             etherstub_lopts, NULL)) != -1) {
5329                 switch (option) {
5330                 case 't':
5331                         flags &= ~DLADM_OPT_PERSIST;
5332                         break;
5333                 case 'R':
5334                         altroot = optarg;
5335                         break;
5336                 default:
5337                         die_opterr(optopt, option, use);
5338                 }
5339         }

5341         /* the etherstub id is the required operand */
5342         if (optind != (argc - 1))
5343                 usage();

5345         if (strlcpy(name, argv[optind], MAXLINKNAMELEN) >= MAXLINKNAMELEN)
5346                 die("link name too long '%s'", argv[optind]);

5348         if (!dladm_valid_linkname(name))
5349                 die("invalid link name '%s'", argv[optind]);

5351         if (altroot != NULL)
5352                 altroot_cmd(altroot, argc, argv);
```



```
5354         status = dladm_vnic_create(handle, name, DATALINK_INVALID_LINKID,
5355             VNIC_MAC_ADDR_TYPE_AUTO, mac_addr, ETHERADDRL, NULL, 0, 0,
5356             VRRP_VRID_NONE, AF_UNSPEC, NULL, NULL, flags);
5357         if (status != DLADM_STATUS_OK)
5358                 die_dlerr(status, "etherstub creation failed");
5359 }

5361 static void
5362 do_delete_etherstub(int argc, char *argv[], const char *use)
5363 {
5364         do_delete_vnic_common(argc, argv, use, B_TRUE);
5365 }

5367 /* ARGSUSED */
5368 static void
5369 do_show_etherstub(int argc, char *argv[], const char *use)
5370 {
5371         do_show_vnic_common(argc, argv, use, B_TRUE);
5372 }

5374 /* ARGSUSED */
5375 static void
5376 do_up_simnet(int argc, char *argv[], const char *use)
5377 {
5378         (void) dladm_simnet_up(handle, DATALINK_ALL_LINKID, 0);
5379 }

5381 static void
5382 do_create_simnet(int argc, char *argv[], const char *use)
5383 {
5384         uint32_t flags;
5385         char *altroot = NULL;
5386         char *media = NULL;
5387         uint32_t mtype = DL_ETHER;
5388         int option;
5389         dladm_status_t status;
5390         char name[MAXLINKNAMELEN];

5392         name[0] = '\0';
5393         flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;

5395         opterr = 0;
5396         while ((option = getopt_long(argc, argv, ":tR:m:",
5397             simnet_lopts, NULL)) != -1) {
5398                 switch (option) {
5399                 case 't':
5400                         flags &= ~DLADM_OPT_PERSIST;
5401                         break;
5402                 case 'R':
5403                         altroot = optarg;
5404                         break;
5405                 case 'm':
5406                         media = optarg;
5407                         break;
5408                 default:
5409                         die_opterr(optopt, option, use);
5410                 }
5411         }

5413         /* the simnet id is the required operand */
5414         if (optind != (argc - 1))
5415                 usage();

5417         if (strlcpy(name, argv[optind], MAXLINKNAMELEN) >= MAXLINKNAMELEN)
5418                 die("link name too long '%s'", argv[optind]);
```

```
5420         if (!dladm_valid_linkname(name))
5421                 die("invalid link name '%s'", name);

5423         if (media != NULL) {
5424                 mtype = dladm_str2media(media);
5425                 if (mtype != DL_ETHER && mtype != DL_WIFI)
5426                         die("media type '%s' is not supported", media);
5427         }

5429         if (altroot != NULL)
5430                 altroot_cmd(altroot, argc, argv);

5432         status = dladm_simnet_create(handle, name, mtype, flags);
5433         if (status != DLADM_STATUS_OK)
5434                 die_dlerr(status, "simnet creation failed");
5435 }

5437 static void
5438 do_delete_simnet(int argc, char *argv[], const char *use)
5439 {
5440         int option;
5441         uint32_t flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
5442         datalink_id_t linkid;
5443         char *altroot = NULL;
5444         dladm_status_t status;
5445         dladm_simnet_attr_t slinfo;

5447         opterr = 0;
5448         while ((option = getopt_long(argc, argv, ":tR:", simnet_lopts,
5449             NULL)) != -1) {
5450                 switch (option) {
5451                 case 't':
5452                         flags &= ~DLADM_OPT_PERSIST;
5453                         break;
5454                 case 'R':
5455                         altroot = optarg;
5456                         break;
5457                 default:
5458                         die_opterr(optopt, option, use);
5459                 }
5460         }

5462         /* get simnet name (required last argument) */
5463         if (optind != (argc - 1))
5464                 usage();

5466         if (!dladm_valid_linkname(argv[optind]))
5467                 die("invalid link name '%s'", argv[optind]);

5469         if (altroot != NULL)
5470                 altroot_cmd(altroot, argc, argv);

5472         status = dladm_name2info(handle, argv[optind], &linkid, NULL, NULL,
5473             NULL);
5474         if (status != DLADM_STATUS_OK)
5475                 die("simnet '%s' not found", argv[optind]);

5477         if ((status = dladm_simnet_info(handle, linkid, &slinfo,
5478             flags)) != DLADM_STATUS_OK)
5479                 die_dlerr(status, "failed to retrieve simnet information");

5481         status = dladm_simnet_delete(handle, linkid, flags);
5482         if (status != DLADM_STATUS_OK)
5483                 die_dlerr(status, "simnet deletion failed");
5484 }
```



```
5486 static void
5487 do_modify_simnet(int argc, char *argv[], const char *use)
5488 {
5489         int option;
5490         uint32_t flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
5491         datalink_id_t linkid;
5492         datalink_id_t peer_linkid;
5493         char *altroot = NULL;
5494         dladm_status_t status;
5495         boolean_t p_arg = B_FALSE;

5497         opterr = 0;
5498         while ((option = getopt_long(argc, argv, ":tR:p:", simnet_lopts,
5499             NULL)) != -1) {
5500                 switch (option) {
5501                 case 't':
5502                         flags &= ~DLADM_OPT_PERSIST;
5503                         break;
5504                 case 'R':
5505                         altroot = optarg;
5506                         break;
5507                 case 'p':
5508                         if (p_arg)
5509                                 die_optdup(option);
5510                         p_arg = B_TRUE;
5511                         if (strcasecmp(optarg, "none") == 0)
5512                                 peer_linkid = DATALINK_INVALID_LINKID;
5513                         else if (dladm_name2info(handle, optarg, &peer_linkid,
5514                             NULL, NULL, NULL) != DLADM_STATUS_OK)
5515                                 die("invalid peer link name '%s'", optarg);
5516                         break;
5517                 default:
5518                         die_opterr(optopt, option, use);
5519                 }
5520         }

5522         /* get simnet name (required last argument) */
5523         if (optind != (argc - 1))
5524                 usage();

5526         /* Nothing to do if no peer link argument */
5527         if (!p_arg)
5528                 return;

5530         if (altroot != NULL)
5531                 altroot_cmd(altroot, argc, argv);

5533         status = dladm_name2info(handle, argv[optind], &linkid, NULL, NULL,
5534             NULL);
5535         if (status != DLADM_STATUS_OK)
5536                 die("invalid link name '%s'", argv[optind]);

5538         status = dladm_simnet_modify(handle, linkid, peer_linkid, flags);
5539         if (status != DLADM_STATUS_OK)
5540                 die_dlerr(status, "simnet modification failed");
5541 }

5543 static dladm_status_t
5544 print_simnet(show_state_t *state, datalink_id_t linkid)
5545 {
5546         dladm_simnet_attr_t     slinfo;
5547         uint32_t                flags;
5548         dladm_status_t          status;
5549         simnet_fields_buf_t     slbuf;
5550         char                    mstr[ETHERADDRL * 3];
```

```
5552         bzero(&slbuf, sizeof (slbuf));
5553         if ((status = dladm_datalink_id2info(handle, linkid, &flags, NULL, NULL,
5554             slbuf.simnet_name, sizeof (slbuf.simnet_name)))
5555             != DLADM_STATUS_OK)
5556                 return (status);

5558         if (!(state->ls_flags & flags))
5559                 return (DLADM_STATUS_NOTFOUND);

5561         if ((status = dladm_simnet_info(handle, linkid, &slinfo,
5562             state->ls_flags)) != DLADM_STATUS_OK)
5563                 return (status);

5565         if (slinfo.sna_peer_link_id != DATALINK_INVALID_LINKID &&
5566             (status = dladm_datalink_id2info(handle, slinfo.sna_peer_link_id,
5567             NULL, NULL, NULL, slbuf.simnet_otherlink,
5568             sizeof (slbuf.simnet_otherlink))) !=
5569             DLADM_STATUS_OK)
5570                 return (status);

5572         if (slinfo.sna_mac_len > sizeof (slbuf.simnet_macaddr))
5573                 return (DLADM_STATUS_BADVAL);

5575         (void) strlcpy(slbuf.simnet_macaddr,
5576             dladm_aggr_macaddr2str(slinfo.sna_mac_addr, mstr),
5577             sizeof (slbuf.simnet_macaddr));
5578         (void) dladm_media2str(slinfo.sna_type, slbuf.simnet_media);

5580         ofmt_print(state->ls_ofmt, &slbuf);
5581         return (status);
5582 }

5584 /* ARGSUSED */
5585 static int
5586 show_simnet(dladm_handle_t dh, datalink_id_t linkid, void *arg)
5587 {
5588         show_state_t            *state = arg;

5590         state->ls_status = print_simnet(state, linkid);
5591         return (DLADM_WALK_CONTINUE);
5592 }

5594 static void
5595 do_show_simnet(int argc, char *argv[], const char *use)
5596 {
5597         int             option;
5598         uint32_t        flags = DLADM_OPT_ACTIVE;
5599         boolean_t       p_arg = B_FALSE;
5600         datalink_id_t   linkid = DATALINK_ALL_LINKID;
5601         show_state_t    state;
5602         dladm_status_t  status;
5603         boolean_t       o_arg = B_FALSE;
5604         ofmt_handle_t   ofmt;
5605         ofmt_status_t   oferr;
5606         char            *all_fields = "link,media,macaddress,otherlink";
5607         char            *fields_str = all_fields;
5608         uint_t          ofmtflags = 0;

5610         bzero(&state, sizeof (state));

5612         opterr = 0;
5613         while ((option = getopt_long(argc, argv, ":pPo:",
5614             show_lopts, NULL)) != -1) {
5615                 switch (option) {
5616                 case 'p':
```



```
5617                         if (p_arg)
5618                                 die_optdup(option);

5620                         p_arg = B_TRUE;
5621                         state.ls_parsable = p_arg;
5622                         break;
5623                 case 'P':
5624                         if (flags != DLADM_OPT_ACTIVE)
5625                                 die_optdup(option);

5627                         flags = DLADM_OPT_PERSIST;
5628                         break;
5629                 case 'o':
5630                         o_arg = B_TRUE;
5631                         fields_str = optarg;
5632                         break;
5633                 default:
5634                         die_opterr(optopt, option, use);
5635                         break;
5636                 }
5637         }

5639         if (p_arg && !o_arg)
5640                 die("-p requires -o");

5642         if (strcasecmp(fields_str, "all") == 0) {
5643                 if (p_arg)
5644                         die("\"-o all\" is invalid with -p");
5645                 fields_str = all_fields;
5646         }

5648         /* get link name (optional last argument) */
5649         if (optind == (argc-1)) {
5650                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
5651                     NULL, NULL, NULL)) != DLADM_STATUS_OK) {
5652                         die_dlerr(status, "link %s is not valid", argv[optind]);
5653                 }
5654         } else if (optind != argc) {
5655                 usage();
5656         }

5658         state.ls_flags = flags;
5659         state.ls_donefirst = B_FALSE;
5660         if (state.ls_parsable)
5661                 ofmtflags |= OFMT_PARSABLE;
5662         oferr = ofmt_open(fields_str, simnet_fields, ofmtflags, 0, &ofmt);
5663         dladm_ofmt_check(oferr, state.ls_parsable, ofmt);
5664         state.ls_ofmt = ofmt;

5666         if (linkid == DATALINK_ALL_LINKID) {
5667                 (void) dladm_walk_datalink_id(show_simnet, handle, &state,
5668                     DATALINK_CLASS_SIMNET, DATALINK_ANY_MEDIATYPE, flags);
5669         } else {
5670                 (void) show_simnet(handle, linkid, &state);
5671                 if (state.ls_status != DLADM_STATUS_OK) {
5672                         ofmt_close(ofmt);
5673                         die_dlerr(state.ls_status, "failed to show simnet %s",
5674                             argv[optind]);
5675                 }
5676         }
5677         ofmt_close(ofmt);
5678 }

5680 static void
5681 link_stats(datalink_id_t linkid, uint_t interval, char *fields_str,
5682     show_state_t *state)
```

```
5683 {
5684         ofmt_handle_t   ofmt;
5685         ofmt_status_t   oferr;
5686         uint_t          ofmtflags = 0;
5687
5688         if (state->ls_parsable)
5689                 ofmtflags |= OFMT_PARSABLE;
5690         oferr = ofmt_open(fields_str, link_s_fields, ofmtflags, 0, &ofmt);
5691         dladm_ofmt_check(oferr, state->ls_parsable, ofmt);
5692         state->ls_ofmt = ofmt;
5693
5694         /*
5695          * If an interval is specified, continuously show the stats
5696          * only for the first MAC port.
5697          */
5698         state->ls_firstonly = (interval != 0);
5699
5700         for (;;) {
5701                 state->ls_donefirst = B_FALSE;
5702                 if (linkid == DATALINK_ALL_LINKID) {
5703                         (void) dladm_walk_datalink_id(show_link_stats, handle,
5704                             state, DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE,
5705                             DLADM_OPT_ACTIVE);
5706                 } else {
5707                         (void) show_link_stats(handle, linkid, state);
5708                 }
5709
5710                 if (interval == 0)
5711                         break;
5712
5713                 (void) fflush(stdout);
5714                 (void) sleep(interval);
5715         }
5716         ofmt_close(ofmt);
5717 }
5718
5719 static void
5720 aggr_stats(datalink_id_t linkid, show_grp_state_t *state, uint_t interval)
5721 {
5722         /*
5723          * If an interval is specified, continuously show the stats
5724          * only for the first group.
5725          */
5726         state->gs_firstonly = (interval != 0);
5727
5728         for (;;) {
5729                 state->gs_donefirst = B_FALSE;
5730                 if (linkid == DATALINK_ALL_LINKID)
5731                         (void) dladm_walk_datalink_id(show_aggr, handle, state,
5732                             DATALINK_CLASS_AGGR, DATALINK_ANY_MEDIATYPE,
5733                             DLADM_OPT_ACTIVE);
5734                 else
5735                         (void) show_aggr(handle, linkid, state);
5736
5737                 if (interval == 0)
5738                         break;
5739
5740                 (void) fflush(stdout);
5741                 (void) sleep(interval);
5742         }
5743 }
5744
5745 /* ARGSUSED */
5746 static void
5747 vnic_stats(show_vnic_state_t *sp, uint32_t interval)
5748 {
```



```
5749         show_vnic_state_t       state;
5750         boolean_t               specific_link, specific_dev;
5751
5752         /* Display vnic statistics */
5753         dump_vnics_head(sp->vs_link);
5754
5755         bzero(&state, sizeof (state));
5756         state.vs_stats = B_TRUE;
5757         state.vs_vnic_id = sp->vs_vnic_id;
5758         state.vs_link_id = sp->vs_link_id;
5759
5760         /*
5761          * If an interval is specified, and a vnic ID is not specified,
5762          * continuously show the stats only for the first vnic.
5763          */
5764         specific_link = (sp->vs_vnic_id != DATALINK_ALL_LINKID);
5765         specific_dev = (sp->vs_link_id != DATALINK_ALL_LINKID);
5766
5767         for (;;) {
5768                 /* Get stats for each vnic */
5769                 state.vs_found = B_FALSE;
5770                 state.vs_donefirst = B_FALSE;
5771                 state.vs_printstats = B_FALSE;
5772                 state.vs_flags = DLADM_OPT_ACTIVE;
5773
5774                 if (!specific_link) {
5775                         (void) dladm_walk_datalink_id(show_vnic, handle, &state,
5776                             DATALINK_CLASS_VNIC, DATALINK_ANY_MEDIATYPE,
5777                             DLADM_OPT_ACTIVE);
5778                 } else {
5779                         (void) show_vnic(handle, sp->vs_vnic_id, &state);
5780                         if (state.vs_status != DLADM_STATUS_OK) {
5781                                 die_dlerr(state.vs_status,
5782                                     "failed to show vnic '%s'", sp->vs_vnic);
5783                         }
5784                 }
5785
5786                 if (specific_link && !state.vs_found)
5787                         die("non-existent vnic '%s'", sp->vs_vnic);
5788                 if (specific_dev && !state.vs_found)
5789                         die("device %s has no vnics", sp->vs_link);
5790
5791                 /* Show totals */
5792                 if ((specific_link | specific_dev) && !interval) {
5793                         (void) printf("Total");
5794                         (void) printf("\t%-10llu",
5795                             state.vs_totalstats.ipackets);
5796                         (void) printf("%-12llu",
5797                             state.vs_totalstats.rbytes);
5798                         (void) printf("%-10llu",
5799                             state.vs_totalstats.opackets);
5800                         (void) printf("%-12llu\n",
5801                             state.vs_totalstats.obytes);
5802                 }
5803
5804                 /* Show stats for each vnic */
5805                 state.vs_donefirst = B_FALSE;
5806                 state.vs_printstats = B_TRUE;
5807
5808                 if (!specific_link) {
5809                         (void) dladm_walk_datalink_id(show_vnic, handle, &state,
5810                             DATALINK_CLASS_VNIC, DATALINK_ANY_MEDIATYPE,
5811                             DLADM_OPT_ACTIVE);
5812                 } else {
5813                         (void) show_vnic(handle, sp->vs_vnic_id, &state);
5814                         if (state.vs_status != DLADM_STATUS_OK) {
```

```
5815                                die_dlerr(state.vs_status,
5816                                    "failed to show vnic '%s'", sp->vs_vnic);
5817                        }
5818                }
5819
5820                if (interval == 0)
5821                        break;
5822
5823                (void) fflush(stdout);
5824                (void) sleep(interval);
5825        }
5826 }
5827
5828 static void
5829 get_mac_stats(const char *dev, pktsum_t *stats)
5830 {
5831        kstat_ctl_t     *kcp;
5832        kstat_t         *ksp;
5833        char module[DLPI_LINKNAME_MAX];
5834        uint_t instance;
5835
5836
5837        bzero(stats, sizeof (*stats));
5838
5839        if (dlpi_parselink(dev, module, &instance) != DLPI_SUCCESS)
5840                return;
5841
5842        if ((kcp = kstat_open()) == NULL) {
5843                warn("kstat open operation failed");
5844                return;
5845        }
5846
5847        ksp = dladm_kstat_lookup(kcp, module, instance, "mac", NULL);
5848        if (ksp != NULL)
5849                dladm_get_stats(kcp, ksp, stats);
5850
5851        (void) kstat_close(kcp);
5852
5853 }
5854
5855 static void
5856 get_link_stats(const char *link, pktsum_t *stats)
5857 {
5858        kstat_ctl_t     *kcp;
5859        kstat_t         *ksp;
5860
5861        bzero(stats, sizeof (*stats));
5862
5863        if ((kcp = kstat_open()) == NULL) {
5864                warn("kstat_open operation failed");
5865                return;
5866        }
5867
5868        ksp = dladm_kstat_lookup(kcp, "link", 0, link, NULL);
5869
5870        if (ksp != NULL)
5871                dladm_get_stats(kcp, ksp, stats);
5872
5873        (void) kstat_close(kcp);
5874 }
5875
5876 static int
5877 query_kstat(char *module, int instance, const char *name, const char *stat,
5878     uint8_t type, void *val)
5879 {
5880        kstat_ctl_t     *kcp;
```



```
5881        kstat_t         *ksp;
5882
5883        if ((kcp = kstat_open()) == NULL) {
5884                warn("kstat open operation failed");
5885                return (-1);
5886        }
5887
5888        if ((ksp = kstat_lookup(kcp, module, instance, (char *)name)) == NULL) {
5889                /*
5890                 * The kstat query could fail if the underlying MAC
5891                 * driver was already detached.
5892                 */
5893                goto bail;
5894        }
5895
5896        if (kstat_read(kcp, ksp, NULL) == -1) {
5897                warn("kstat read failed");
5898                goto bail;
5899        }
5900
5901        if (dladm_kstat_value(ksp, stat, type, val) < 0)
5902                goto bail;
5903
5904        (void) kstat_close(kcp);
5905        return (0);
5906
5907 bail:
5908        (void) kstat_close(kcp);
5909        return (-1);
5910 }
5911
5912 static int
5913 get_one_kstat(const char *name, const char *stat, uint8_t type,
5914     void *val, boolean_t islink)
5915 {
5916        char            module[DLPI_LINKNAME_MAX];
5917        uint_t          instance;
5918
5919        if (islink) {
5920                return (query_kstat("link", 0, name, stat, type, val));
5921        } else {
5922                if (dlpi_parselink(name, module, &instance) != DLPI_SUCCESS)
5923                        return (-1);
5924
5925                return (query_kstat(module, instance, "mac", stat, type, val));
5926        }
5927 }
5928
5929 static uint64_t
5930 get_ifspeed(const char *name, boolean_t islink)
5931 {
5932        uint64_t ifspeed = 0;
5933
5934        (void) get_one_kstat(name, "ifspeed", KSTAT_DATA_UINT64,
5935            &ifspeed, islink);
5936
5937        return (ifspeed);
5938 }
5939
5940 static const char *
5941 get_linkstate(const char *name, boolean_t islink, char *buf)
5942 {
5943        link_state_t    linkstate;
5944
5945        if (get_one_kstat(name, "link_state", KSTAT_DATA_UINT32,
5946            &linkstate, islink) != 0) {
```

```
5947                 (void) strlcpy(buf, "?", DLADM_STRSIZE);
5948                 return (buf);
5949         }
5950         return (dladm_linkstate2str(linkstate, buf));
5951 }

5953 static const char *
5954 get_linkduplex(const char *name, boolean_t islink, char *buf)
5955 {
5956         link_duplex_t   linkduplex;

5958         if (get_one_kstat(name, "link_duplex", KSTAT_DATA_UINT32,
5959             &linkduplex, islink) != 0) {
5960                 (void) strlcpy(buf, "unknown", DLADM_STRSIZE);
5961                 return (buf);
5962         }

5964         return (dladm_linkduplex2str(linkduplex, buf));
5965 }

5967 static int
5968 parse_wifi_fields(char *str, ofmt_handle_t *ofmt, uint_t cmdtype,
5969     boolean_t parsable)
5970 {
5971         ofmt_field_t    *template, *of;
5972         ofmt_cb_t       *fn;
5973         ofmt_status_t   oferr;

5975         if (cmdtype == WIFI_CMD_SCAN) {
5976                 template = wifi_common_fields;
5977                 if (str == NULL)
5978                         str = def_scan_wifi_fields;
5979                 if (strcasecmp(str, "all") == 0)
5980                         str = all_scan_wifi_fields;
5981                 fn = print_wlan_attr_cb;
5982         } else if (cmdtype == WIFI_CMD_SHOW) {
5983                 bcopy(wifi_common_fields, &wifi_show_fields[2],
5984                     sizeof (wifi_common_fields));
5985                 template = wifi_show_fields;
5986                 if (str == NULL)
5987                         str = def_show_wifi_fields;
5988                 if (strcasecmp(str, "all") == 0)
5989                         str = all_show_wifi_fields;
5990                 fn = print_link_attr_cb;
5991         } else {
5992                 return (-1);
5993         }

5995         for (of = template; of->of_name != NULL; of++) {
5996                 if (of->of_cb == NULL)
5997                         of->of_cb = fn;
5998         }

6000         oferr = ofmt_open(str, template, (parsable ? OFMT_PARSABLE : 0),
6001             0, ofmt);
6002         dladm_ofmt_check(oferr, parsable, *ofmt);
6003         return (0);
6004 }

6006 typedef struct print_wifi_state {
6007         char            *ws_link;
6008         boolean_t       ws_parsable;
6009         boolean_t       ws_header;
6010         ofmt_handle_t   ws_ofmt;
6011 } print_wifi_state_t;
```



```
6013 typedef struct  wlan_scan_args_s {
6014         print_wifi_state_t      *ws_state;
6015         void                    *ws_attr;
6016 } wlan_scan_args_t;

6018 static boolean_t
6019 print_wlan_attr_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
6020 {
6021         wlan_scan_args_t        *w = ofarg->ofmt_cbarg;
6022         print_wifi_state_t      *statep = w->ws_state;
6023         dladm_wlan_attr_t       *attrp = w->ws_attr;
6024         char                    tmpbuf[DLADM_STRSIZE];

6026         if (ofarg->ofmt_id == 0) {
6027                 (void) strlcpy(buf, (char *)statep->ws_link, bufsize);
6028                 return (B_TRUE);
6029         }

6031         if ((ofarg->ofmt_id & attrp->wa_valid) == 0)
6032                 return (B_TRUE);

6034         switch (ofarg->ofmt_id) {
6035         case DLADM_WLAN_ATTR_ESSID:
6036                 (void) dladm_wlan_essid2str(&attrp->wa_essid, tmpbuf);
6037                 break;
6038         case DLADM_WLAN_ATTR_BSSID:
6039                 (void) dladm_wlan_bssid2str(&attrp->wa_bssid, tmpbuf);
6040                 break;
6041         case DLADM_WLAN_ATTR_SECMODE:
6042                 (void) dladm_wlan_secmode2str(&attrp->wa_secmode, tmpbuf);
6043                 break;
6044         case DLADM_WLAN_ATTR_STRENGTH:
6045                 (void) dladm_wlan_strength2str(&attrp->wa_strength, tmpbuf);
6046                 break;
6047         case DLADM_WLAN_ATTR_MODE:
6048                 (void) dladm_wlan_mode2str(&attrp->wa_mode, tmpbuf);
6049                 break;
6050         case DLADM_WLAN_ATTR_SPEED:
6051                 (void) dladm_wlan_speed2str(&attrp->wa_speed, tmpbuf);
6052                 (void) strlcat(tmpbuf, "Mb", sizeof (tmpbuf));
6053                 break;
6054         case DLADM_WLAN_ATTR_AUTH:
6055                 (void) dladm_wlan_auth2str(&attrp->wa_auth, tmpbuf);
6056                 break;
6057         case DLADM_WLAN_ATTR_BSSTYPE:
6058                 (void) dladm_wlan_bsstype2str(&attrp->wa_bsstype, tmpbuf);
6059                 break;
6060         }
6061         (void) strlcpy(buf, tmpbuf, bufsize);

6063         return (B_TRUE);
6064 }

6066 static boolean_t
6067 print_scan_results(void *arg, dladm_wlan_attr_t *attrp)
6068 {
6069         print_wifi_state_t      *statep = arg;
6070         wlan_scan_args_t        warg;

6072         bzero(&warg, sizeof (warg));
6073         warg.ws_state = statep;
6074         warg.ws_attr = attrp;
6075         ofmt_print(statep->ws_ofmt, &warg);
6076         return (B_TRUE);
6077 }
```

```
6079 static int
6080 scan_wifi(dladm_handle_t dh, datalink_id_t linkid, void *arg)
6081 {
6082         print_wifi_state_t      *statep = arg;
6083         dladm_status_t          status;
6084         char                    link[MAXLINKNAMELEN];

6086         if ((status = dladm_datalink_id2info(dh, linkid, NULL, NULL, NULL, link,
6087             sizeof (link))) != DLADM_STATUS_OK) {
6088                 return (DLADM_WALK_CONTINUE);
6089         }

6091         statep->ws_link = link;
6092         status = dladm_wlan_scan(dh, linkid, statep, print_scan_results);
6093         if (status != DLADM_STATUS_OK)
6094                 die_dlerr(status, "cannot scan link '%s'", statep->ws_link);

6096         return (DLADM_WALK_CONTINUE);
6097 }

6099 static boolean_t
6100 print_wifi_status_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
6101 {
6102         static char             tmpbuf[DLADM_STRSIZE];
6103         wlan_scan_args_t        *w = ofarg->ofmt_cbarg;
6104         dladm_wlan_linkattr_t   *attrp = w->ws_attr;

6106         if ((ofarg->ofmt_id & attrp->la_valid) != 0) {
6107                 (void) dladm_wlan_linkstatus2str(&attrp->la_status, tmpbuf);
6108                 (void) strlcpy(buf, tmpbuf, bufsize);
6109         }
6110         return (B_TRUE);
6111 }

6113 static boolean_t
6114 print_link_attr_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
6115 {
6116         wlan_scan_args_t        *w = ofarg->ofmt_cbarg, w1;
6117         print_wifi_state_t      *statep = w->ws_state;
6118         dladm_wlan_linkattr_t   *attrp = w->ws_attr;

6120         bzero(&w1, sizeof (w1));
6121         w1.ws_state = statep;
6122         w1.ws_attr = &attrp->la_wlan_attr;
6123         ofarg->ofmt_cbarg = &w1;
6124         return (print_wlan_attr_cb(ofarg, buf, bufsize));
6125 }

6127 static int
6128 show_wifi(dladm_handle_t dh, datalink_id_t linkid, void *arg)
6129 {
6130         print_wifi_state_t      *statep = arg;
6131         dladm_wlan_linkattr_t   attr;
6132         dladm_status_t          status;
6133         char                    link[MAXLINKNAMELEN];
6134         wlan_scan_args_t        warg;

6136         if ((status = dladm_datalink_id2info(dh, linkid, NULL, NULL, NULL, link,
6137             sizeof (link))) != DLADM_STATUS_OK) {
6138                 return (DLADM_WALK_CONTINUE);
6139         }

6141         /* dladm_wlan_get_linkattr() memsets attr with 0 */
6142         status = dladm_wlan_get_linkattr(dh, linkid, &attr);
6143         if (status != DLADM_STATUS_OK)
6144                 die_dlerr(status, "cannot get link attributes for %s", link);
```



```
6146         statep->ws_link = link;

6148         bzero(&warg, sizeof (warg));
6149         warg.ws_state = statep;
6150         warg.ws_attr = &attr;
6151         ofmt_print(statep->ws_ofmt, &warg);
6152         return (DLADM_WALK_CONTINUE);
6153 }

6155 static void
6156 do_display_wifi(int argc, char **argv, int cmd, const char *use)
6157 {
6158         int                     option;
6159         char                    *fields_str = NULL;
6160         int             (*callback)(dladm_handle_t, datalink_id_t, void *);
6161         print_wifi_state_t      state;
6162         datalink_id_t           linkid = DATALINK_ALL_LINKID;
6163         dladm_status_t          status;

6165         if (cmd == WIFI_CMD_SCAN)
6166                 callback = scan_wifi;
6167         else if (cmd == WIFI_CMD_SHOW)
6168                 callback = show_wifi;
6169         else
6170                 return;

6172         state.ws_parsable = B_FALSE;
6173         state.ws_header = B_TRUE;
6174         opterr = 0;
6175         while ((option = getopt_long(argc, argv, ":o:p",
6176             wifi_longopts, NULL)) != -1) {
6177                 switch (option) {
6178                 case 'o':
6179                         fields_str = optarg;
6180                         break;
6181                 case 'p':
6182                         state.ws_parsable = B_TRUE;
6183                         break;
6184                 default:
6185                         die_opterr(optopt, option, use);
6186                 }
6187         }

6189         if (state.ws_parsable && fields_str == NULL)
6190                 die("-p requires -o");

6192         if (state.ws_parsable && strcasecmp(fields_str, "all") == 0)
6193                 die("\"-o all\" is invalid with -p");

6195         if (optind == (argc - 1)) {
6196                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
6197                     NULL, NULL, NULL)) != DLADM_STATUS_OK) {
6198                         die_dlerr(status, "link %s is not valid", argv[optind]);
6199                 }
6200         } else if (optind != argc) {
6201                 usage();
6202         }

6204         if (parse_wifi_fields(fields_str, &state.ws_ofmt, cmd,
6205             state.ws_parsable) < 0)
6206                 die("invalid field(s) specified");

6208         if (linkid == DATALINK_ALL_LINKID) {
6209                 (void) dladm_walk_datalink_id(callback, handle, &state,
6210                     DATALINK_CLASS_PHYS | DATALINK_CLASS_SIMNET,
```

```
6211                    DL_WIFI, DLADM_OPT_ACTIVE);
6212        } else {
6213                (void) (*callback)(handle, linkid, &state);
6214        }
6215        ofmt_close(state.ws_ofmt);
6216 }

6218 static void
6219 do_scan_wifi(int argc, char **argv, const char *use)
6220 {
6221        do_display_wifi(argc, argv, WIFI_CMD_SCAN, use);
6222 }

6224 static void
6225 do_show_wifi(int argc, char **argv, const char *use)
6226 {
6227        do_display_wifi(argc, argv, WIFI_CMD_SHOW, use);
6228 }

6230 typedef struct wlan_count_attr {
6231        uint_t          wc_count;
6232        datalink_id_t   wc_linkid;
6233 } wlan_count_attr_t;

6235 /* ARGSUSED */
6236 static int
6237 do_count_wlan(dladm_handle_t dh, datalink_id_t linkid, void *arg)
6238 {
6239        wlan_count_attr_t *cp = arg;

6241        if (cp->wc_count == 0)
6242                cp->wc_linkid = linkid;
6243        cp->wc_count++;
6244        return (DLADM_WALK_CONTINUE);
6245 }

6247 static int
6248 parse_wlan_keys(char *str, dladm_wlan_key_t **keys, uint_t *key_countp)
6249 {
6250        uint_t                  i;
6251        dladm_wlan_key_t        *wk;
6252        int                     nfields = 1;
6253        char                    *field, *token, *lasts = NULL, c;

6255        token = str;
6256        while ((c = *token++) != NULL) {
6257                if (c == ',')
6258                        nfields++;
6259        }
6260        token = strdup(str);
6261        if (token == NULL)
6262                return (-1);

6264        wk = malloc(nfields * sizeof (dladm_wlan_key_t));
6265        if (wk == NULL)
6266                goto fail;

6268        token = str;
6269        for (i = 0; i < nfields; i++) {
6270                char                    *s;
6271                dladm_secobj_class_t    class;
6272                dladm_status_t          status;

6274                field = strtok_r(token, ",", &lasts);
6275                token = NULL;
```



```
6277                (void) strlcpy(wk[i].wk_name, field,
6278                    DLADM_WLAN_MAX_KEYNAME_LEN);

6280                wk[i].wk_idx = 1;
6281                if ((s = strrchr(wk[i].wk_name, ':')) != NULL) {
6282                        if (s[1] == '\0' || s[2] != '\0' || !isdigit(s[1]))
6283                                goto fail;

6285                        wk[i].wk_idx = (uint_t)(s[1] - '0');
6286                        *s = '\0';
6287                }
6288                wk[i].wk_len = DLADM_WLAN_MAX_KEY_LEN;

6290                status = dladm_get_secobj(handle, wk[i].wk_name, &class,
6291                    wk[i].wk_val, &wk[i].wk_len, 0);
6292                if (status != DLADM_STATUS_OK) {
6293                        if (status == DLADM_STATUS_NOTFOUND) {
6294                                status = dladm_get_secobj(handle, wk[i].wk_name,
6295                                    &class, wk[i].wk_val, &wk[i].wk_len,
6296                                    DLADM_OPT_PERSIST);
6297                        }
6298                        if (status != DLADM_STATUS_OK)
6299                                goto fail;
6300                }
6301                wk[i].wk_class = class;
6302        }
6303        *keys = wk;
6304        *key_countp = i;
6305        free(token);
6306        return (0);
6307 fail:
6308        free(wk);
6309        free(token);
6310        return (-1);
6311 }

6313 static void
6314 do_connect_wifi(int argc, char **argv, const char *use)
6315 {
6316        int                     option;
6317        dladm_wlan_attr_t       attr, *attrp;
6318        dladm_status_t          status = DLADM_STATUS_OK;
6319        int                     timeout = DLADM_WLAN_CONNECT_TIMEOUT_DEFAULT;
6320        datalink_id_t           linkid = DATALINK_ALL_LINKID;
6321        dladm_wlan_key_t        *keys = NULL;
6322        uint_t                  key_count = 0;
6323        uint_t                  flags = 0;
6324        dladm_wlan_secmode_t    keysecmode = DLADM_WLAN_SECMODE_NONE;
6325        char                    buf[DLADM_STRSIZE];

6327        opterr = 0;
6328        (void) memset(&attr, 0, sizeof (attr));
6329        while ((option = getopt_long(argc, argv, ":e:i:a:m:b:s:k:T:c",
6330            wifi_longopts, NULL)) != -1) {
6331                switch (option) {
6332                case 'e':
6333                        status = dladm_wlan_str2essid(optarg, &attr.wa_essid);
6334                        if (status != DLADM_STATUS_OK)
6335                                die("invalid ESSID '%s'", optarg);

6337                        attr.wa_valid |= DLADM_WLAN_ATTR_ESSID;
6338                        /*
6339                         * Try to connect without doing a scan.
6340                         */
6341                        flags |= DLADM_WLAN_CONNECT_NOSCAN;
6342                        break;
```

```
6343                case 'i':
6344                        status = dladm_wlan_str2bssid(optarg, &attr.wa_bssid);
6345                        if (status != DLADM_STATUS_OK)
6346                                die("invalid BSSID %s", optarg);
6347
6348                        attr.wa_valid |= DLADM_WLAN_ATTR_BSSID;
6349                        break;
6350                case 'a':
6351                        status = dladm_wlan_str2auth(optarg, &attr.wa_auth);
6352                        if (status != DLADM_STATUS_OK)
6353                                die("invalid authentication mode '%s'", optarg);
6354
6355                        attr.wa_valid |= DLADM_WLAN_ATTR_AUTH;
6356                        break;
6357                case 'm':
6358                        status = dladm_wlan_str2mode(optarg, &attr.wa_mode);
6359                        if (status != DLADM_STATUS_OK)
6360                                die("invalid mode '%s'", optarg);
6361
6362                        attr.wa_valid |= DLADM_WLAN_ATTR_MODE;
6363                        break;
6364                case 'b':
6365                        if ((status = dladm_wlan_str2bsstype(optarg,
6366                            &attr.wa_bsstype)) != DLADM_STATUS_OK) {
6367                                die("invalid bsstype '%s'", optarg);
6368                        }
6369
6370                        attr.wa_valid |= DLADM_WLAN_ATTR_BSSTYPE;
6371                        break;
6372                case 's':
6373                        if ((status = dladm_wlan_str2secmode(optarg,
6374                            &attr.wa_secmode)) != DLADM_STATUS_OK) {
6375                                die("invalid security mode '%s'", optarg);
6376                        }
6377
6378                        attr.wa_valid |= DLADM_WLAN_ATTR_SECMODE;
6379                        break;
6380                case 'k':
6381                        if (parse_wlan_keys(optarg, &keys, &key_count) < 0)
6382                                die("invalid key(s) '%s'", optarg);
6383
6384                        if (keys[0].wk_class == DLADM_SECOBJ_CLASS_WEP)
6385                                keysecmode = DLADM_WLAN_SECMODE_WEP;
6386                        else
6387                                keysecmode = DLADM_WLAN_SECMODE_WPA;
6388                        break;
6389                case 'T':
6390                        if (strcasecmp(optarg, "forever") == 0) {
6391                                timeout = -1;
6392                                break;
6393                        }
6394                        if (!str2int(optarg, &timeout) || timeout < 0)
6395                                die("invalid timeout value '%s'", optarg);
6396                        break;
6397                case 'c':
6398                        flags |= DLADM_WLAN_CONNECT_CREATEIBSS;
6399                        flags |= DLADM_WLAN_CONNECT_CREATEIBSS;
6400                        break;
6401                default:
6402                        die_opterr(optopt, option, use);
6403                        break;
6404                }
6405        }
6406
6407        if (keysecmode == DLADM_WLAN_SECMODE_NONE) {
6408                if ((attr.wa_valid & DLADM_WLAN_ATTR_SECMODE) != 0) {
```



```
6409                        die("key required for security mode '%s'",
6410                            dladm_wlan_secmode2str(&attr.wa_secmode, buf));
6411                }
6412        } else {
6413                if ((attr.wa_valid & DLADM_WLAN_ATTR_SECMODE) != 0 &&
6414                    attr.wa_secmode != keysecmode)
6415                        die("incompatible -s and -k options");
6416                attr.wa_valid |= DLADM_WLAN_ATTR_SECMODE;
6417                attr.wa_secmode = keysecmode;
6418        }
6419
6420        if (optind == (argc - 1)) {
6421                if ((status = dladm_name2info(handle, argv[optind], &linkid,
6422                    NULL, NULL, NULL)) != DLADM_STATUS_OK) {
6423                        die_dlerr(status, "link %s is not valid", argv[optind]);
6424                }
6425        } else if (optind != argc) {
6426                usage();
6427        }
6428
6429        if (linkid == DATALINK_ALL_LINKID) {
6430                wlan_count_attr_t wcattr;
6431
6432                wcattr.wc_linkid = DATALINK_INVALID_LINKID;
6433                wcattr.wc_count = 0;
6434                (void) dladm_walk_datalink_id(do_count_wlan, handle, &wcattr,
6435                    DATALINK_CLASS_PHYS | DATALINK_CLASS_SIMNET,
6436                    DL_WIFI, DLADM_OPT_ACTIVE);
6437                if (wcattr.wc_count == 0) {
6438                        die("no wifi links are available");
6439                } else if (wcattr.wc_count > 1) {
6440                        die("link name is required when more than one wifi "
6441                            "link is available");
6442                }
6443                linkid = wcattr.wc_linkid;
6444        }
6445        attrp = (attr.wa_valid == 0) ? NULL : &attr;
6446 again:
6447        if ((status = dladm_wlan_connect(handle, linkid, attrp, timeout, keys,
6448            key_count, flags)) != DLADM_STATUS_OK) {
6449                if ((flags & DLADM_WLAN_CONNECT_NOSCAN) != 0) {
6450                        /*
6451                         * Try again with scanning and filtering.
6452                         */
6453                        flags &= ~DLADM_WLAN_CONNECT_NOSCAN;
6454                        goto again;
6455                }
6456
6457                if (status == DLADM_STATUS_NOTFOUND) {
6458                        if (attr.wa_valid == 0) {
6459                                die("no wifi networks are available");
6460                        } else {
6461                                die("no wifi networks with the specified "
6462                                    "criteria are available");
6463                        }
6464                }
6465                die_dlerr(status, "cannot connect");
6466        }
6467        free(keys);
6468 }
6469
6470 /* ARGSUSED */
6471 static int
6472 do_all_disconnect_wifi(dladm_handle_t dh, datalink_id_t linkid, void *arg)
6473 {
6474        dladm_status_t  status;
```

```
6476         status = dladm_wlan_disconnect(dh, linkid);
6477         if (status != DLADM_STATUS_OK)
6478                 warn_dlerr(status, "cannot disconnect link");

6480         return (DLADM_WALK_CONTINUE);
6481 }

6483 static void
6484 do_disconnect_wifi(int argc, char **argv, const char *use)
6485 {
6486         int                     option;
6487         datalink_id_t           linkid = DATALINK_ALL_LINKID;
6488         boolean_t               all_links = B_FALSE;
6489         dladm_status_t          status;
6490         wlan_count_attr_t       wcattr;

6492         opterr = 0;
6493         while ((option = getopt_long(argc, argv, ":a",
6494             wifi_longopts, NULL)) != -1) {
6495                 switch (option) {
6496                 case 'a':
6497                         all_links = B_TRUE;
6498                         break;
6499                 default:
6500                         die_opterr(optopt, option, use);
6501                         break;
6502                 }
6503         }

6505         if (optind == (argc - 1)) {
6506                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
6507                     NULL, NULL, NULL)) != DLADM_STATUS_OK) {
6508                         die_dlerr(status, "link %s is not valid", argv[optind]);
6509                 }
6510         } else if (optind != argc) {
6511                 usage();
6512         }

6514         if (linkid == DATALINK_ALL_LINKID) {
6515                 if (!all_links) {
6516                         wcattr.wc_linkid = linkid;
6517                         wcattr.wc_count = 0;
6518                         (void) dladm_walk_datalink_id(do_count_wlan, handle,
6519                             &wcattr,
6520                             DATALINK_CLASS_PHYS | DATALINK_CLASS_SIMNET,
6521                             DL_WIFI, DLADM_OPT_ACTIVE);
6522                         if (wcattr.wc_count == 0) {
6523                                 die("no wifi links are available");
6524                         } else if (wcattr.wc_count > 1) {
6525                                 die("link name is required when more than "
6526                                     "one wifi link is available");
6527                         }
6528                         linkid = wcattr.wc_linkid;
6529                 } else {
6530                         (void) dladm_walk_datalink_id(do_all_disconnect_wifi,
6531                             handle, NULL,
6532                             DATALINK_CLASS_PHYS | DATALINK_CLASS_SIMNET,
6533                             DL_WIFI, DLADM_OPT_ACTIVE);
6534                         return;
6535                 }
6536         }
6537         status = dladm_wlan_disconnect(handle, linkid);
6538         if (status != DLADM_STATUS_OK)
6539                 die_dlerr(status, "cannot disconnect");
6540 }
```



```
6542 static void
6543 print_linkprop(datalink_id_t linkid, show_linkprop_state_t *statep,
6544     const char *propname, dladm_prop_type_t type, const char *format,
6545     char **pptr)
6546 {
6547         int             i;
6548         char            *ptr, *lim;
6549         char            buf[DLADM_STRSIZE];
6550         char            *unknown = "--", *notsup = "";
6551         char            **propvals = statep->ls_propvals;
6552         uint_t          valcnt = DLADM_MAX_PROP_VALCNT;
6553         dladm_status_t  status;

6555         status = dladm_get_linkprop(handle, linkid, type, propname, propvals,
6556             &valcnt);
6557         if (status != DLADM_STATUS_OK) {
6558                 if (status == DLADM_STATUS_TEMPONLY) {
6559                         if (type == DLADM_PROP_VAL_MODIFIABLE &&
6560                             statep->ls_persist) {
6561                                 valcnt = 1;
6562                                 propvals = &unknown;
6563                         } else {
6564                                 statep->ls_status = status;
6565                                 statep->ls_retstatus = status;
6566                                 return;
6567                         }
6568                 } else if (status == DLADM_STATUS_NOTSUP ||
6569                     statep->ls_persist) {
6570                         valcnt = 1;
6571                         if (type == DLADM_PROP_VAL_CURRENT ||
6572                             type == DLADM_PROP_VAL_PERM)
6573                                 propvals = &unknown;
6574                         else
6575                                 propvals = &notsup;
6576                 } else if (status == DLADM_STATUS_NOTDEFINED) {
6577                         propvals = &notsup; /* STR_UNDEF_VAL */
6578                 } else {
6579                         if (statep->ls_proplist &&
6580                             statep->ls_status == DLADM_STATUS_OK) {
6581                                 warn_dlerr(status,
6582                                     "cannot get link property '%s' for %s",
6583                                     propname, statep->ls_link);
6584                         }
6585                         statep->ls_status = status;
6586                         statep->ls_retstatus = status;
6587                         return;
6588                 }
6589         }

6591         statep->ls_status = DLADM_STATUS_OK;

6593         buf[0] = '\0';
6594         ptr = buf;
6595         lim = buf + DLADM_STRSIZE;
6596         for (i = 0; i < valcnt; i++) {
6597                 if (propvals[i][0] == '\0' && !statep->ls_parsable)
6598                         ptr += snprintf(ptr, lim - ptr, "--,");
6599                 else
6600                         ptr += snprintf(ptr, lim - ptr, "%s,", propvals[i]);
6601                 if (ptr >= lim)
6602                         break;
6603         }
6604         if (valcnt > 0)
6605                 buf[strlen(buf) - 1] = '\0';
```

```
6607         lim = statep->ls_line + MAX_PROP_LINE;
6608         if (statep->ls_parsable) {
6609                 *pptr += snprintf(*pptr, lim - *pptr,
6610                     "%s", buf);
6611         } else {
6612                 *pptr += snprintf(*pptr, lim - *pptr, format, buf);
6613         }
6614 }

6616 static boolean_t
6617 print_linkprop_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
6618 {
6619         linkprop_args_t         *arg = ofarg->ofmt_cbarg;
6620         char                    *propname = arg->ls_propname;
6621         show_linkprop_state_t   *statep = arg->ls_state;
6622         char                    *ptr = statep->ls_line;
6623         char                    *lim = ptr + MAX_PROP_LINE;
6624         datalink_id_t           linkid = arg->ls_linkid;

6626         switch (ofarg->ofmt_id) {
6627         case LINKPROP_LINK:
6628                 (void) snprintf(ptr, lim - ptr, "%s", statep->ls_link);
6629                 break;
6630         case LINKPROP_PROPERTY:
6631                 (void) snprintf(ptr, lim - ptr, "%s", propname);
6632                 break;
6633         case LINKPROP_VALUE:
6634                 print_linkprop(linkid, statep, propname,
6635                     statep->ls_persist ? DLADM_PROP_VAL_PERSISTENT :
6636                     DLADM_PROP_VAL_CURRENT, "%s", &ptr);
6637                 /*
6638                  * If we failed to query the link property, for example, query
6639                  * the persistent value of a non-persistable link property,
6640                  * simply skip the output.
6641                  */
6642                 if (statep->ls_status != DLADM_STATUS_OK) {
6643                         /*
6644                          * Ignore the temponly error when we skip printing
6645                          * link properties to avoid returning failure on exit.
6646                          */
6647                         if (statep->ls_retstatus == DLADM_STATUS_TEMPONLY)
6648                                 statep->ls_retstatus = DLADM_STATUS_OK;
6649                         goto skip;
6650                 }
6651                 ptr = statep->ls_line;
6652                 break;
6653         case LINKPROP_PERM:
6654                 print_linkprop(linkid, statep, propname,
6655                     DLADM_PROP_VAL_PERM, "%s", &ptr);
6656                 if (statep->ls_status != DLADM_STATUS_OK)
6657                         goto skip;
6658                 ptr = statep->ls_line;
6659                 break;
6660         case LINKPROP_DEFAULT:
6661                 print_linkprop(linkid, statep, propname,
6662                     DLADM_PROP_VAL_DEFAULT, "%s", &ptr);
6663                 if (statep->ls_status != DLADM_STATUS_OK)
6664                         goto skip;
6665                 ptr = statep->ls_line;
6666                 break;
6667         case LINKPROP_POSSIBLE:
6668                 print_linkprop(linkid, statep, propname,
6669                     DLADM_PROP_VAL_MODIFIABLE, "%s ", &ptr);
6670                 if (statep->ls_status != DLADM_STATUS_OK)
6671                         goto skip;
6672                 ptr = statep->ls_line;
```



```
6673                 break;
6674         default:
6675                 die("invalid input");
6676                 break;
6677         }
6678         (void) strlcpy(buf, ptr, bufsize);
6679         return (B_TRUE);
6680 skip:
6681         return ((statep->ls_status == DLADM_STATUS_OK) ?
6682             B_TRUE : B_FALSE);
6683 }

6685 static boolean_t
6686 linkprop_is_supported(datalink_id_t  linkid, const char *propname,
6687     show_linkprop_state_t *statep)
6688 {
6689         dladm_status_t  status;
6690         uint_t          valcnt = DLADM_MAX_PROP_VALCNT;

6692         /* if used with -p flag, always print output */
6693         if (statep->ls_proplist != NULL)
6694                 return (B_TRUE);

6696         status = dladm_get_linkprop(handle, linkid, DLADM_PROP_VAL_DEFAULT,
6697             propname, statep->ls_propvals, &valcnt);

6699         if (status == DLADM_STATUS_OK)
6700                 return (B_TRUE);

6702         /*
6703          * A system wide default value is not available for the
6704          * property. Check if current value can be retrieved.
6705          */
6706         status = dladm_get_linkprop(handle, linkid, DLADM_PROP_VAL_CURRENT,
6707             propname, statep->ls_propvals, &valcnt);

6709         return (status == DLADM_STATUS_OK);
6710 }

6712 /* ARGSUSED */
6713 static int
6714 show_linkprop(dladm_handle_t dh, datalink_id_t linkid, const char *propname,
6715     void *arg)
6716 {
6717         show_linkprop_state_t   *statep = arg;
6718         linkprop_args_t         ls_arg;

6720         bzero(&ls_arg, sizeof (ls_arg));
6721         ls_arg.ls_state = statep;
6722         ls_arg.ls_propname = (char *)propname;
6723         ls_arg.ls_linkid = linkid;

6725         /*
6726          * This will need to be fixed when kernel interfaces are added
6727          * to enable walking of all known private properties. For now,
6728          * we are limited to walking persistent private properties only.
6729          */
6730         if ((propname[0] == '_') && !statep->ls_persist &&
6731             (statep->ls_proplist == NULL))
6732                 return (DLADM_WALK_CONTINUE);
6733         if (!statep->ls_parsable &&
6734             !linkprop_is_supported(linkid, propname, statep))
6735                 return (DLADM_WALK_CONTINUE);

6737         ofmt_print(statep->ls_ofmt, &ls_arg);
```

```
6739         return (DLADM_WALK_CONTINUE);
6740 }

6742 static void
6743 do_show_linkprop(int argc, char **argv, const char *use)
6744 {
6745         int                     option;
6746         char                    propstr[DLADM_STRSIZE];
6747         dladm_arg_list_t        *proplist = NULL;
6748         datalink_id_t           linkid = DATALINK_ALL_LINKID;
6749         show_linkprop_state_t   state;
6750         uint32_t                flags = DLADM_OPT_ACTIVE;
6751         dladm_status_t          status;
6752         char                    *fields_str = NULL;
6753         ofmt_handle_t           ofmt;
6754         ofmt_status_t           oferr;
6755         uint_t                  ofmtflags = 0;

6757         bzero(propstr, DLADM_STRSIZE);
6758         opterr = 0;
6759         state.ls_propvals = NULL;
6760         state.ls_line = NULL;
6761         state.ls_parsable = B_FALSE;
6762         state.ls_persist = B_FALSE;
6763         state.ls_header = B_TRUE;
6764         state.ls_retstatus = DLADM_STATUS_OK;

6766         while ((option = getopt_long(argc, argv, ":p:cPo:",
6767             prop_longopts, NULL)) != -1) {
6768                 switch (option) {
6769                 case 'p':
6770                         (void) strlcat(propstr, optarg, DLADM_STRSIZE);
6771                         if (strlcat(propstr, ",", DLADM_STRSIZE) >=
6772                             DLADM_STRSIZE)
6773                                 die("property list too long '%s'", propstr);
6774                         break;
6775                 case 'c':
6776                         state.ls_parsable = B_TRUE;
6777                         break;
6778                 case 'P':
6779                         state.ls_persist = B_TRUE;
6780                         flags = DLADM_OPT_PERSIST;
6781                         break;
6782                 case 'o':
6783                         fields_str = optarg;
6784                         break;
6785                 default:
6786                         die_opterr(optopt, option, use);
6787                         break;
6788                 }
6789         }

6791         if (optind == (argc - 1)) {
6792                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
6793                     NULL, NULL, NULL)) != DLADM_STATUS_OK) {
6794                         die_dlerr(status, "link %s is not valid", argv[optind]);
6795                 }
6796         } else if (optind != argc) {
6797                 usage();
6798         }

6800         if (dladm_parse_link_props(propstr, &proplist, B_TRUE)
6801             != DLADM_STATUS_OK)
6802                 die("invalid link properties specified");
6803         state.ls_proplist = proplist;
6804         state.ls_status = DLADM_STATUS_OK;
```



```
6806         if (state.ls_parsable)
6807                 ofmtflags |= OFMT_PARSABLE;
6808         else
6809                 ofmtflags |= OFMT_WRAP;

6811         oferr = ofmt_open(fields_str, linkprop_fields, ofmtflags, 0, &ofmt);
6812         dladm_ofmt_check(oferr, state.ls_parsable, ofmt);
6813         state.ls_ofmt = ofmt;

6815         if (linkid == DATALINK_ALL_LINKID) {
6816                 (void) dladm_walk_datalink_id(show_linkprop_onelink, handle,
6817                     &state, DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE, flags);
6818         } else {
6819                 (void) show_linkprop_onelink(handle, linkid, &state);
6820         }
6821         ofmt_close(ofmt);
6822         dladm_free_props(proplist);

6824         if (state.ls_retstatus != DLADM_STATUS_OK) {
6825                 dladm_close(handle);
6826                 exit(EXIT_FAILURE);
6827         }
6828 }

6830 static int
6831 show_linkprop_onelink(dladm_handle_t hdl, datalink_id_t linkid, void *arg)
6832 {
6833         int                     i;
6834         char                    *buf;
6835         uint32_t                flags;
6836         dladm_arg_list_t        *proplist = NULL;
6837         show_linkprop_state_t   *statep = arg;
6838         dlpi_handle_t           dh = NULL;

6840         statep->ls_status = DLADM_STATUS_OK;

6842         if (dladm_datalink_id2info(hdl, linkid, &flags, NULL, NULL,
6843             statep->ls_link, MAXLINKNAMELEN) != DLADM_STATUS_OK) {
6844                 statep->ls_status = DLADM_STATUS_NOTFOUND;
6845                 return (DLADM_WALK_CONTINUE);
6846         }

6848         if ((statep->ls_persist && !(flags & DLADM_OPT_PERSIST)) ||
6849             (!statep->ls_persist && !(flags & DLADM_OPT_ACTIVE))) {
6850                 statep->ls_status = DLADM_STATUS_BADARG;
6851                 return (DLADM_WALK_CONTINUE);
6852         }

6854         proplist = statep->ls_proplist;

6856         /*
6857          * When some WiFi links are opened for the first time, their hardware
6858          * automatically scans for APs and does other slow operations.  Thus,
6859          * if there are no open links, the retrieval of link properties
6860          * (below) will proceed slowly unless we hold the link open.
6861          *
6862          * Note that failure of dlpi_open() does not necessarily mean invalid
6863          * link properties, because dlpi_open() may fail because of incorrect
6864          * autopush configuration. Therefore, we ingore the return value of
6865          * dlpi_open().
6866          */
6867         if (!statep->ls_persist)
6868                 (void) dlpi_open(statep->ls_link, &dh, 0);

6870         buf = malloc((sizeof (char *) + DLADM_PROP_VAL_MAX) *
```

```
6871            DLADM_MAX_PROP_VALCNT + MAX_PROP_LINE);
6872        if (buf == NULL)
6873                die("insufficient memory");

6875        statep->ls_propvals = (char **)(void *)buf;
6876        for (i = 0; i < DLADM_MAX_PROP_VALCNT; i++) {
6877                statep->ls_propvals[i] = buf +
6878                    sizeof (char *) * DLADM_MAX_PROP_VALCNT +
6879                    i * DLADM_PROP_VAL_MAX;
6880        }
6881        statep->ls_line = buf +
6882            (sizeof (char *) + DLADM_PROP_VAL_MAX) * DLADM_MAX_PROP_VALCNT;

6884        if (proplist != NULL) {
6885                for (i = 0; i < proplist->al_count; i++) {
6886                        (void) show_linkprop(hdl, linkid,
6887                            proplist->al_info[i].ai_name, statep);
6888                }
6889        } else {
6890                (void) dladm_walk_linkprop(hdl, linkid, statep,
6891                    show_linkprop);
6892        }
6893        if (dh != NULL)
6894                dlpi_close(dh);
6895        free(buf);
6896        return (DLADM_WALK_CONTINUE);
6897 }

6899 static int
6900 reset_one_linkprop(dladm_handle_t dh, datalink_id_t linkid,
6901     const char *propname, void *arg)
6902 {
6903        set_linkprop_state_t    *statep = arg;
6904        dladm_status_t          status;

6906        status = dladm_set_linkprop(dh, linkid, propname, NULL, 0,
6907            DLADM_OPT_ACTIVE | (statep->ls_temp ? 0 : DLADM_OPT_PERSIST));
6908        if (status != DLADM_STATUS_OK &&
6909            status != DLADM_STATUS_PROPRDONLY &&
6910            status != DLADM_STATUS_NOTSUP) {
6911                warn_dlerr(status, "cannot reset link property '%s' on '%s'",
6912                    propname, statep->ls_name);
6913                statep->ls_status = status;
6914        }

6916        return (DLADM_WALK_CONTINUE);
6917 }

6919 static void
6920 set_linkprop(int argc, char **argv, boolean_t reset, const char *use)
6921 {
6922        int                     i, option;
6923        char                    errmsg[DLADM_STRSIZE];
6924        char                    *altroot = NULL;
6925        datalink_id_t           linkid;
6926        boolean_t               temp = B_FALSE;
6927        dladm_status_t          status = DLADM_STATUS_OK;
6928        char                    propstr[DLADM_STRSIZE];
6929        dladm_arg_list_t        *proplist = NULL;

6931        opterr = 0;
6932        bzero(propstr, DLADM_STRSIZE);

6934        while ((option = getopt_long(argc, argv, ":p:R:t",
6935            prop_longopts, NULL)) != -1) {
6936                switch (option) {
```



```
6937                case 'p':
6938                        (void) strlcat(propstr, optarg, DLADM_STRSIZE);
6939                        if (strlcat(propstr, ",", DLADM_STRSIZE) >=
6940                            DLADM_STRSIZE)
6941                                die("property list too long '%s'", propstr);
6942                        break;
6943                case 't':
6944                        temp = B_TRUE;
6945                        break;
6946                case 'R':
6947                        altroot = optarg;
6948                        break;
6949                default:
6950                        die_opterr(optopt, option, use);

6952                }
6953        }

6955        /* get link name (required last argument) */
6956        if (optind != (argc - 1))
6957                usage();

6959        if (dladm_parse_link_props(propstr, &proplist, reset) !=
6960            DLADM_STATUS_OK)
6961                die("invalid link properties specified");

6963        if (proplist == NULL && !reset)
6964                die("link property must be specified");

6966        if (altroot != NULL) {
6967                dladm_free_props(proplist);
6968                altroot_cmd(altroot, argc, argv);
6969        }

6971        status = dladm_name2info(handle, argv[optind], &linkid, NULL, NULL,
6972            NULL);
6973        if (status != DLADM_STATUS_OK)
6974                die_dlerr(status, "link %s is not valid", argv[optind]);

6976        if (proplist == NULL) {
6977                set_linkprop_state_t    state;

6979                state.ls_name = argv[optind];
6980                state.ls_reset = reset;
6981                state.ls_temp = temp;
6982                state.ls_status = DLADM_STATUS_OK;

6984                (void) dladm_walk_linkprop(handle, linkid, &state,
6985                    reset_one_linkprop);

6987                status = state.ls_status;
6988                goto done;
6989        }

6991        for (i = 0; i < proplist->al_count; i++) {
6992                dladm_arg_info_t        *aip = &proplist->al_info[i];
6993                char            **val;
6994                uint_t          count;

6996                if (reset) {
6997                        val = NULL;
6998                        count = 0;
6999                } else {
7000                        val = aip->ai_val;
7001                        count = aip->ai_count;
7002                        if (count == 0) {
```

```
7003                                warn("no value specified for '%s'",
7004                                    aip->ai_name);
7005                                status = DLADM_STATUS_BADARG;
7006                                continue;
7007                        }
7008                }
7009                status = dladm_set_linkprop(handle, linkid, aip->ai_name, val,
7010                    count, DLADM_OPT_ACTIVE | (temp ? 0 : DLADM_OPT_PERSIST));
7011                switch (status) {
7012                case DLADM_STATUS_OK:
7013                        break;
7014                case DLADM_STATUS_NOTFOUND:
7015                        warn("invalid link property '%s'", aip->ai_name);
7016                        break;
7017                case DLADM_STATUS_BADVAL: {
7018                        int             j;
7019                        char            *ptr, *lim;
7020                        char            **propvals = NULL;
7021                        uint_t          valcnt = DLADM_MAX_PROP_VALCNT;
7022                        dladm_status_t  s;

7024                        ptr = malloc((sizeof (char *) +
7025                            DLADM_PROP_VAL_MAX) * DLADM_MAX_PROP_VALCNT +
7026                            MAX_PROP_LINE);

7028                        propvals = (char **)(void *)ptr;
7029                        if (propvals == NULL)
7030                                die("insufficient memory");

7032                        for (j = 0; j < DLADM_MAX_PROP_VALCNT; j++) {
7033                                propvals[j] = ptr + sizeof (char *) *
7034                                    DLADM_MAX_PROP_VALCNT +
7035                                    j * DLADM_PROP_VAL_MAX;
7036                        }
7037                        s = dladm_get_linkprop(handle, linkid,
7038                            DLADM_PROP_VAL_MODIFIABLE, aip->ai_name, propvals,
7039                            &valcnt);

7041                        if (s != DLADM_STATUS_OK) {
7042                                warn_dlerr(status, "cannot set link property "
7043                                    "'%s' on '%s'", aip->ai_name, argv[optind]);
7044                                free(propvals);
7045                                break;
7046                        }

7048                        ptr = errmsg;
7049                        lim = ptr + DLADM_STRSIZE;
7050                        *ptr = '\0';
7051                        for (j = 0; j < valcnt; j++) {
7052                                ptr += snprintf(ptr, lim - ptr, "%s,",
7053                                    propvals[j]);
7054                                if (ptr >= lim)
7055                                        break;
7056                        }
7057                        if (ptr > errmsg) {
7058                                *(ptr - 1) = '\0';
7059                                warn("link property '%s' must be one of: %s",
7060                                    aip->ai_name, errmsg);
7061                        } else
7062                                warn("invalid link property '%s'", *val);
7063                        free(propvals);
7064                        break;
7065                }
7066                default:
7067                        if (reset) {
7068                                warn_dlerr(status, "cannot reset link property "
```



```
7069                                    "'%s' on '%s'", aip->ai_name, argv[optind]);
7070                        } else {
7071                                warn_dlerr(status, "cannot set link property "
7072                                    "'%s' on '%s'", aip->ai_name, argv[optind]);
7073                        }
7074                        break;
7075                }
7076        }
7077 done:
7078        dladm_free_props(proplist);
7079        if (status != DLADM_STATUS_OK) {
7080                dladm_close(handle);
7081                exit(EXIT_FAILURE);
7082        }
7083 }

7085 static void
7086 do_set_linkprop(int argc, char **argv, const char *use)
7087 {
7088        set_linkprop(argc, argv, B_FALSE, use);
7089 }

7091 static void
7092 do_reset_linkprop(int argc, char **argv, const char *use)
7093 {
7094        set_linkprop(argc, argv, B_TRUE, use);
7095 }

7097 static int
7098 convert_secobj(char *buf, uint_t len, uint8_t *obj_val, uint_t *obj_lenp,
7099     dladm_secobj_class_t class)
7100 {
7101        int error = 0;

7103        if (class == DLADM_SECOBJ_CLASS_WPA) {
7104                if (len < 8 || len > 63)
7105                        return (EINVAL);
7106                (void) memcpy(obj_val, buf, len);
7107                *obj_lenp = len;
7108                return (error);
7109        }

7111        if (class == DLADM_SECOBJ_CLASS_WEP) {
7112                switch (len) {
7113                case 5:                 /* ASCII key sizes */
7114                case 13:
7115                        (void) memcpy(obj_val, buf, len);
7116                        *obj_lenp = len;
7117                        break;
7118                case 10:                /* Hex key sizes, not preceded by 0x */
7119                case 26:
7120                        error = hexascii_to_octet(buf, len, obj_val, obj_lenp);
7121                        break;
7122                case 12:                /* Hex key sizes, preceded by 0x */
7123                case 28:
7124                        if (strncmp(buf, "0x", 2) != 0)
7125                                return (EINVAL);
7126                        error = hexascii_to_octet(buf + 2, len - 2,
7127                            obj_val, obj_lenp);
7128                        break;
7129                default:
7130                        return (EINVAL);
7131                }
7132                return (error);
7133        }
```

```
7135         return (ENOENT);
7136 }

7138 static void
7139 defersig(int sig)
7140 {
7141         signalled = sig;
7142 }

7144 static int
7145 get_secobj_from_tty(uint_t try, const char *objname, char *buf)
7146 {
7147         uint_t          len = 0;
7148         int             c;
7149         struct termios  stored, current;
7150         void            (*sigfunc)(int);

7152         /*
7153          * Turn off echo -- but before we do so, defer SIGINT handling
7154          * so that a ^C doesn't leave the terminal corrupted.
7155          */
7156         sigfunc = signal(SIGINT, defersig);
7157         (void) fflush(stdin);
7158         (void) tcgetattr(0, &stored);
7159         current = stored;
7160         current.c_lflag &= ~(ICANON|ECHO);
7161         current.c_cc[VTIME] = 0;
7162         current.c_cc[VMIN] = 1;
7163         (void) tcsetattr(0, TCSANOW, &current);
7164 again:
7165         if (try == 1)
7166                 (void) printf(gettext("provide value for '%s': "), objname);
7167         else
7168                 (void) printf(gettext("confirm value for '%s': "), objname);

7170         (void) fflush(stdout);
7171         while (signalled == 0) {
7172                 c = getchar();
7173                 if (c == '\n' || c == '\r') {
7174                         if (len != 0)
7175                                 break;
7176                         (void) putchar('\n');
7177                         goto again;
7178                 }

7180                 buf[len++] = c;
7181                 if (len >= DLADM_SECOBJ_VAL_MAX - 1)
7182                         break;
7183                 (void) putchar('*');
7184         }

7186         (void) putchar('\n');
7187         (void) fflush(stdin);

7189         /*
7190          * Restore terminal setting and handle deferred signals.
7191          */
7192         (void) tcsetattr(0, TCSANOW, &stored);

7194         (void) signal(SIGINT, sigfunc);
7195         if (signalled != 0)
7196                 (void) kill(getpid(), signalled);

7198         return (len);
7199 }
```



```
7201 static int
7202 get_secobj_val(char *obj_name, uint8_t *obj_val, uint_t *obj_lenp,
7203     dladm_secobj_class_t class, FILE *filep)
7204 {
7205         int             rval;
7206         uint_t          len, len2;
7207         char            buf[DLADM_SECOBJ_VAL_MAX], buf2[DLADM_SECOBJ_VAL_MAX];

7209         if (filep == NULL) {
7210                 len = get_secobj_from_tty(1, obj_name, buf);
7211                 rval = convert_secobj(buf, len, obj_val, obj_lenp, class);
7212                 if (rval == 0) {
7213                         len2 = get_secobj_from_tty(2, obj_name, buf2);
7214                         if (len != len2 || memcmp(buf, buf2, len) != 0)
7215                                 rval = ENOTSUP;
7216                 }
7217                 return (rval);
7218         } else {
7219                 for (;;) {
7220                         if (fgets(buf, sizeof (buf), filep) == NULL)
7221                                 break;
7222                         if (isspace(buf[0]))
7223                                 continue;

7225                         len = strlen(buf);
7226                         if (buf[len - 1] == '\n') {
7227                                 buf[len - 1] = '\0';
7228                                 len--;
7229                         }
7230                         break;
7231                 }
7232                 (void) fclose(filep);
7233         }
7234         return (convert_secobj(buf, len, obj_val, obj_lenp, class));
7235 }

7237 static boolean_t
7238 check_auth(const char *auth)
7239 {
7240         struct passwd   *pw;

7242         if ((pw = getpwuid(getuid())) == NULL)
7243                 return (B_FALSE);

7245         return (chkauthattr(auth, pw->pw_name) != 0);
7246 }

7248 static void
7249 audit_secobj(char *auth, char *class, char *obj,
7250     boolean_t success, boolean_t create)
7251 {
7252         adt_session_data_t      *ah;
7253         adt_event_data_t        *event;
7254         au_event_t              flag;
7255         char                    *errstr;

7257         if (create) {
7258                 flag = ADT_dladm_create_secobj;
7259                 errstr = "ADT_dladm_create_secobj";
7260         } else {
7261                 flag = ADT_dladm_delete_secobj;
7262                 errstr = "ADT_dladm_delete_secobj";
7263         }

7265         if (adt_start_session(&ah, NULL, ADT_USE_PROC_DATA) != 0)
7266                 die("adt_start_session: %s", strerror(errno));
```

```
7268         if ((event = adt_alloc_event(ah, flag)) == NULL)
7269                 die("adt_alloc_event (%s): %s", errstr, strerror(errno));

7271         /* fill in audit info */
7272         if (create) {
7273                 event->adt_dladm_create_secobj.auth_used = auth;
7274                 event->adt_dladm_create_secobj.obj_class = class;
7275                 event->adt_dladm_create_secobj.obj_name = obj;
7276         } else {
7277                 event->adt_dladm_delete_secobj.auth_used = auth;
7278                 event->adt_dladm_delete_secobj.obj_class = class;
7279                 event->adt_dladm_delete_secobj.obj_name = obj;
7280         }

7282         if (success) {
7283                 if (adt_put_event(event, ADT_SUCCESS, ADT_SUCCESS) != 0) {
7284                         die("adt_put_event (%s, success): %s", errstr,
7285                             strerror(errno));
7286                 }
7287         } else {
7288                 if (adt_put_event(event, ADT_FAILURE,
7289                     ADT_FAIL_VALUE_AUTH) != 0) {
7290                         die("adt_put_event: (%s, failure): %s", errstr,
7291                             strerror(errno));
7292                 }
7293         }

7295         adt_free_event(event);
7296         (void) adt_end_session(ah);
7297 }

7299 static void
7300 do_create_secobj(int argc, char **argv, const char *use)
7301 {
7302         int                     option, rval;
7303         FILE                    *filep = NULL;
7304         char                    *obj_name = NULL;
7305         char                    *class_name = NULL;
7306         uint8_t                 obj_val[DLADM_SECOBJ_VAL_MAX];
7307         uint_t                  obj_len;
7308         boolean_t               success, temp = B_FALSE;
7309         dladm_status_t          status;
7310         dladm_secobj_class_t    class = -1;
7311         uid_t                   euid;

7313         opterr = 0;
7314         (void) memset(obj_val, 0, DLADM_SECOBJ_VAL_MAX);
7315         while ((option = getopt_long(argc, argv, ":f:c:R:t",
7316             wifi_longopts, NULL)) != -1) {
7317                 switch (option) {
7318                 case 'f':
7319                         euid = geteuid();
7320                         (void) seteuid(getuid());
7321                         filep = fopen(optarg, "r");
7322                         if (filep == NULL) {
7323                                 die("cannot open %s: %s", optarg,
7324                                     strerror(errno));
7325                         }
7326                         (void) seteuid(euid);
7327                         break;
7328                 case 'c':
7329                         class_name = optarg;
7330                         status = dladm_str2secobjclass(optarg, &class);
7331                         if (status != DLADM_STATUS_OK) {
7332                                 die("invalid secure object class '%s', "
```



```
7333                                     "valid values are: wep, wpa", optarg);
7334                         }
7335                         break;
7336                 case 't':
7337                         temp = B_TRUE;
7338                         break;
7339                 case 'R':
7340                         status = dladm_set_rootdir(optarg);
7341                         if (status != DLADM_STATUS_OK) {
7342                                 die_dlerr(status, "invalid directory "
7343                                     "specified");
7344                         }
7345                         break;
7346                 default:
7347                         die_opterr(optopt, option, use);
7348                         break;
7349                 }
7350         }

7352         if (optind == (argc - 1))
7353                 obj_name = argv[optind];
7354         else if (optind != argc)
7355                 usage();

7357         if (class == -1)
7358                 die("secure object class required");

7360         if (obj_name == NULL)
7361                 die("secure object name required");

7363         if (!dladm_valid_secobj_name(obj_name))
7364                 die("invalid secure object name '%s'", obj_name);

7366         success = check_auth(LINK_SEC_AUTH);
7367         audit_secobj(LINK_SEC_AUTH, class_name, obj_name, success, B_TRUE);
7368         if (!success)
7369                 die("authorization '%s' is required", LINK_SEC_AUTH);

7371         rval = get_secobj_val(obj_name, obj_val, &obj_len, class, filep);
7372         if (rval != 0) {
7373                 switch (rval) {
7374                 case ENOENT:
7375                         die("invalid secure object class");
7376                         break;
7377                 case EINVAL:
7378                         die("invalid secure object value");
7379                         break;
7380                 case ENOTSUP:
7381                         die("verification failed");
7382                         break;
7383                 default:
7384                         die("invalid secure object: %s", strerror(rval));
7385                         break;
7386                 }
7387         }

7389         status = dladm_set_secobj(handle, obj_name, class, obj_val, obj_len,
7390             DLADM_OPT_CREATE | DLADM_OPT_ACTIVE);
7391         if (status != DLADM_STATUS_OK) {
7392                 die_dlerr(status, "could not create secure object '%s'",
7393                     obj_name);
7394         }
7395         if (temp)
7396                 return;

7398         status = dladm_set_secobj(handle, obj_name, class, obj_val, obj_len,
```

```
7399             DLADM_OPT_PERSIST);
7400         if (status != DLADM_STATUS_OK) {
7401                 warn_dlerr(status, "could not persistently create secure "
7402                     "object '%s'", obj_name);
7403         }
7404 }

7406 static void
7407 do_delete_secobj(int argc, char **argv, const char *use)
7408 {
7409         int             i, option;
7410         boolean_t       temp = B_FALSE;
7411         boolean_t       success;
7412         dladm_status_t  status, pstatus;
7413         int             nfields = 1;
7414         char            *field, *token, *lasts = NULL, c;

7416         opterr = 0;
7417         status = pstatus = DLADM_STATUS_OK;
7418         while ((option = getopt_long(argc, argv, ":R:t",
7419             wifi_longopts, NULL)) != -1) {
7420                 switch (option) {
7421                 case 't':
7422                         temp = B_TRUE;
7423                         break;
7424                 case 'R':
7425                         status = dladm_set_rootdir(optarg);
7426                         if (status != DLADM_STATUS_OK) {
7427                                 die_dlerr(status, "invalid directory "
7428                                     "specified");
7429                         }
7430                         break;
7431                 default:
7432                         die_opterr(optopt, option, use);
7433                         break;
7434                 }
7435         }

7437         if (optind != (argc - 1))
7438                 die("secure object name required");

7440         token = argv[optind];
7441         while ((c = *token++) != NULL) {
7442                 if (c == ',')
7443                         nfields++;
7444         }
7445         token = strdup(argv[optind]);
7446         if (token == NULL)
7447                 die("no memory");

7449         success = check_auth(LINK_SEC_AUTH);
7450         audit_secobj(LINK_SEC_AUTH, "unknown", argv[optind], success, B_FALSE);
7451         if (!success)
7452                 die("authorization '%s' is required", LINK_SEC_AUTH);

7454         for (i = 0; i < nfields; i++) {

7456                 field = strtok_r(token, ",", &lasts);
7457                 token = NULL;
7458                 status = dladm_unset_secobj(handle, field, DLADM_OPT_ACTIVE);
7459                 if (!temp) {
7460                         pstatus = dladm_unset_secobj(handle, field,
7461                             DLADM_OPT_PERSIST);
7462                 } else {
7463                         pstatus = DLADM_STATUS_OK;
7464                 }
```



```
7466                 if (status != DLADM_STATUS_OK) {
7467                         warn_dlerr(status, "could not delete secure object "
7468                             "'%s'", field);
7469                 }
7470                 if (pstatus != DLADM_STATUS_OK) {
7471                         warn_dlerr(pstatus, "could not persistently delete "
7472                             "secure object '%s'", field);
7473                 }
7474         }
7475         free(token);

7477         if (status != DLADM_STATUS_OK || pstatus != DLADM_STATUS_OK) {
7478                 dladm_close(handle);
7479                 exit(EXIT_FAILURE);
7480         }
7481 }

7483 typedef struct show_secobj_state {
7484         boolean_t       ss_persist;
7485         boolean_t       ss_parsable;
7486         boolean_t       ss_header;
7487         ofmt_handle_t   ss_ofmt;
7488 } show_secobj_state_t;


7491 static boolean_t
7492 show_secobj(dladm_handle_t dh, void *arg, const char *obj_name)
7493 {
7494         uint_t                  obj_len = DLADM_SECOBJ_VAL_MAX;
7495         uint8_t                 obj_val[DLADM_SECOBJ_VAL_MAX];
7496         char                    buf[DLADM_STRSIZE];
7497         uint_t                  flags = 0;
7498         dladm_secobj_class_t    class;
7499         show_secobj_state_t     *statep = arg;
7500         dladm_status_t          status;
7501         secobj_fields_buf_t     sbuf;

7503         bzero(&sbuf, sizeof (secobj_fields_buf_t));
7504         if (statep->ss_persist)
7505                 flags |= DLADM_OPT_PERSIST;

7507         status = dladm_get_secobj(dh, obj_name, &class, obj_val, &obj_len,
7508             flags);
7509         if (status != DLADM_STATUS_OK)
7510                 die_dlerr(status, "cannot get secure object '%s'", obj_name);

7512         (void) snprintf(sbuf.ss_obj_name, sizeof (sbuf.ss_obj_name),
7513             obj_name);
7514         (void) dladm_secobjclass2str(class, buf);
7515         (void) snprintf(sbuf.ss_class, sizeof (sbuf.ss_class), "%s", buf);
7516         if (getuid() == 0) {
7517                 char    val[DLADM_SECOBJ_VAL_MAX * 2];
7518                 uint_t  len = sizeof (val);

7520                 if (octet_to_hexascii(obj_val, obj_len, val, &len) == 0)
7521                         (void) snprintf(sbuf.ss_val,
7522                             sizeof (sbuf.ss_val), "%s", val);
7523         }
7524         ofmt_print(statep->ss_ofmt, &sbuf);
7525         return (B_TRUE);
7526 }

7528 static void
7529 do_show_secobj(int argc, char **argv, const char *use)
7530 {
```

```
7531         int                     option;
7532         show_secobj_state_t     state;
7533         dladm_status_t          status;
7534         boolean_t               o_arg = B_FALSE;
7535         uint_t                  i;
7536         uint_t                  flags;
7537         char                    *fields_str = NULL;
7538         char                    *def_fields = "object,class";
7539         char                    *all_fields = "object,class,value";
7540         char                    *field, *token, *lasts = NULL, c;
7541         ofmt_handle_t           ofmt;
7542         ofmt_status_t           oferr;
7543         uint_t                  ofmtflags = 0;

7545         opterr = 0;
7546         bzero(&state, sizeof (state));
7547         state.ss_parsable = B_FALSE;
7548         fields_str = def_fields;
7549         state.ss_persist = B_FALSE;
7550         state.ss_parsable = B_FALSE;
7551         state.ss_header = B_TRUE;
7552         while ((option = getopt_long(argc, argv, ":pPo:",
7553             wifi_longopts, NULL)) != -1) {
7554                 switch (option) {
7555                 case 'p':
7556                         state.ss_parsable = B_TRUE;
7557                         break;
7558                 case 'P':
7559                         state.ss_persist = B_TRUE;
7560                         break;
7561                 case 'o':
7562                         o_arg = B_TRUE;
7563                         if (strcasecmp(optarg, "all") == 0)
7564                                 fields_str = all_fields;
7565                         else
7566                                 fields_str = optarg;
7567                         break;
7568                 default:
7569                         die_opterr(optopt, option, use);
7570                         break;
7571                 }
7572         }

7574         if (state.ss_parsable && !o_arg)
7575                 die("option -c requires -o");

7577         if (state.ss_parsable && fields_str == all_fields)
7578                 die("\"-o all\" is invalid with -p");

7580         if (state.ss_parsable)
7581                 ofmtflags |= OFMT_PARSABLE;
7582         oferr = ofmt_open(fields_str, secobj_fields, ofmtflags, 0, &ofmt);
7583         dladm_ofmt_check(oferr, state.ss_parsable, ofmt);
7584         state.ss_ofmt = ofmt;

7586         flags = state.ss_persist ? DLADM_OPT_PERSIST : 0;

7588         if (optind == (argc - 1)) {
7589                 uint_t obj_fields = 1;

7591                 token = argv[optind];
7592                 if (token == NULL)
7593                         die("secure object name required");
7594                 while ((c = *token++) != NULL) {
7595                         if (c == ',')
7596                                 obj_fields++;
```



```
7597                 }
7598                 token = strdup(argv[optind]);
7599                 if (token == NULL)
7600                         die("no memory");
7601                 for (i = 0; i < obj_fields; i++) {
7602                         field = strtok_r(token, ",", &lasts);
7603                         token = NULL;
7604                         if (!show_secobj(handle, &state, field))
7605                                 break;
7606                 }
7607                 free(token);
7608                 ofmt_close(ofmt);
7609                 return;
7610         } else if (optind != argc)
7611                 usage();

7613         status = dladm_walk_secobj(handle, &state, show_secobj, flags);

7615         if (status != DLADM_STATUS_OK)
7616                 die_dlerr(status, "show-secobj");
7617         ofmt_close(ofmt);
7618 }

7620 /*ARGSUSED*/
7621 static int
7622 i_dladm_init_linkprop(dladm_handle_t dh, datalink_id_t linkid, void *arg)
7623 {
7624         (void) dladm_init_linkprop(dh, linkid, B_TRUE);
7625         return (DLADM_WALK_CONTINUE);
7626 }

7628 /*ARGSUSED*/
7629 void
7630 do_init_linkprop(int argc, char **argv, const char *use)
7631 {
7632         int                     option;
7633         dladm_status_t          status;
7634         datalink_id_t           linkid = DATALINK_ALL_LINKID;
7635         datalink_media_t        media = DATALINK_ANY_MEDIATYPE;
7636         uint_t                  any_media = B_TRUE;

7638         opterr = 0;
7639         while ((option = getopt(argc, argv, ":w")) != -1) {
7640                 switch (option) {
7641                 case 'w':
7642                         media = DL_WIFI;
7643                         any_media = B_FALSE;
7644                         break;
7645                 default:
7646                         /*
7647                          * Because init-linkprop is not a public command,
7648                          * print the usage instead.
7649                          */
7650                         usage();
7651                         break;
7652                 }
7653         }

7655         if (optind == (argc - 1)) {
7656                 if ((status = dladm_name2info(handle, argv[optind], &linkid,
7657                     NULL, NULL, NULL)) != DLADM_STATUS_OK)
7658                         die_dlerr(status, "link %s is not valid", argv[optind]);
7659         } else if (optind != argc) {
7660                 usage();
7661         }
```

```
7663         if (linkid == DATALINK_ALL_LINKID) {
7664                 /*
7665                  * linkprops of links of other classes have been initialized as
7666                  * part of the dladm up-xxx operation.
7667                  */
7668                 (void) dladm_walk_datalink_id(i_dladm_init_linkprop, handle,
7669                     NULL, DATALINK_CLASS_PHYS, media, DLADM_OPT_PERSIST);
7670         } else {
7671                 (void) dladm_init_linkprop(handle, linkid, any_media);
7672         }
7673 }

7675 static void
7676 do_show_ether(int argc, char **argv, const char *use)
7677 {
7678         int                     option;
7679         datalink_id_t           linkid;
7680         print_ether_state_t     state;
7681         char                    *fields_str = NULL;
7682         ofmt_handle_t           ofmt;
7683         ofmt_status_t           oferr;
7684         uint_t                  ofmtflags = 0;

7686         bzero(&state, sizeof (state));
7687         state.es_link = NULL;
7688         state.es_parsable = B_FALSE;

7690         while ((option = getopt_long(argc, argv, "o:px",
7691             showeth_lopts, NULL)) != -1) {
7692                 switch (option) {
7693                         case 'x':
7694                                 state.es_extended = B_TRUE;
7695                                 break;
7696                         case 'p':
7697                                 state.es_parsable = B_TRUE;
7698                                 break;
7699                         case 'o':
7700                                 fields_str = optarg;
7701                                 break;
7702                         default:
7703                                 die_opterr(optopt, option, use);
7704                                 break;
7705                 }
7706         }

7708         if (optind == (argc - 1))
7709                 state.es_link = argv[optind];

7711         if (state.es_parsable)
7712                 ofmtflags |= OFMT_PARSABLE;
7713         oferr = ofmt_open(fields_str, ether_fields, ofmtflags,
7714             DLADM_DEFAULT_COL, &ofmt);
7715         dladm_ofmt_check(oferr, state.es_parsable, ofmt);
7716         state.es_ofmt = ofmt;

7718         if (state.es_link == NULL) {
7719                 (void) dladm_walk_datalink_id(show_etherprop, handle, &state,
7720                     DATALINK_CLASS_PHYS, DL_ETHER, DLADM_OPT_ACTIVE);
7721         } else {
7722                 if (!link_is_ether(state.es_link, &linkid))
7723                         die("invalid link specified");
7724                 (void) show_etherprop(handle, linkid, &state);
7725         }
7726         ofmt_close(ofmt);
7727 }
```



```
7729 static int
7730 show_etherprop(dladm_handle_t dh, datalink_id_t linkid, void *arg)
7731 {
7732         print_ether_state_t     *statep = arg;
7733         ether_fields_buf_t      ebuf;
7734         dladm_ether_info_t      eattr;
7735         dladm_status_t          status;

7737         bzero(&ebuf, sizeof (ether_fields_buf_t));
7738         if (dladm_datalink_id2info(dh, linkid, NULL, NULL, NULL,
7739             ebuf.eth_link, sizeof (ebuf.eth_link)) != DLADM_STATUS_OK) {
7740                 return (DLADM_WALK_CONTINUE);
7741         }

7743         status = dladm_ether_info(dh, linkid, &eattr);
7744         if (status != DLADM_STATUS_OK)
7745                 goto cleanup;

7747         (void) strlcpy(ebuf.eth_ptype, "current", sizeof (ebuf.eth_ptype));

7749         (void) dladm_ether_autoneg2str(ebuf.eth_autoneg,
7750             sizeof (ebuf.eth_autoneg), &eattr, CURRENT);
7751         (void) dladm_ether_pause2str(ebuf.eth_pause,
7752             sizeof (ebuf.eth_pause), &eattr, CURRENT);
7753         (void) dladm_ether_spdx2str(ebuf.eth_spdx,
7754             sizeof (ebuf.eth_spdx), &eattr, CURRENT);
7755         (void) strlcpy(ebuf.eth_state,
7756             dladm_linkstate2str(eattr.lei_state, ebuf.eth_state),
7757             sizeof (ebuf.eth_state));
7758         (void) strlcpy(ebuf.eth_rem_fault,
7759             (eattr.lei_attr[CURRENT].le_fault ? "fault" : "none"),
7760             sizeof (ebuf.eth_rem_fault));

7762         ofmt_print(statep->es_ofmt, &ebuf);

7764         if (statep->es_extended)
7765                 show_ether_xprop(arg, &eattr);

7767 cleanup:
7768         dladm_ether_info_done(&eattr);
7769         return (DLADM_WALK_CONTINUE);
7770 }

7772 /* ARGSUSED */
7773 static void
7774 do_init_secobj(int argc, char **argv, const char *use)
7775 {
7776         dladm_status_t  status;

7778         status = dladm_init_secobj(handle);
7779         if (status != DLADM_STATUS_OK)
7780                 die_dlerr(status, "secure object initialization failed");
7781 }

7783 enum bridge_func {
7784         brCreate, brAdd, brModify
7785 };

7787 static void
7788 create_modify_add_bridge(int argc, char **argv, const char *use,
7789     enum bridge_func func)
7790 {
7791         int                     option;
7792         uint_t                  n, i, nlink;
7793         uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
7794         char                    *altroot = NULL;
```

```
7795        char                    *links[MAXPORT];
7796        datalink_id_t           linkids[MAXPORT];
7797        dladm_status_t          status;
7798        const char              *bridge;
7799        UID_STP_CFG_T           cfg, cfg_old;
7800        dladm_bridge_prot_t     brprot = DLADM_BRIDGE_PROT_UNKNOWN;
7801        dladm_bridge_prot_t     brprot_old;

7803        /* Set up the default configuration values */
7804        cfg.field_mask = 0;
7805        cfg.bridge_priority = DEF_BR_PRIO;
7806        cfg.max_age = DEF_BR_MAXAGE;
7807        cfg.hello_time = DEF_BR_HELLOT;
7808        cfg.forward_delay = DEF_BR_FWDELAY;
7809        cfg.force_version = DEF_FORCE_VERS;

7811        nlink = opterr = 0;
7812        while ((option = getopt_long(argc, argv, ":P:R:d:f:h:l:m:p:",
7813            bridge_lopts, NULL)) != -1) {
7814                switch (option) {
7815                case 'P':
7816                        if (func == brAdd)
7817                                die_opterr(optopt, option, use);
7818                        status = dladm_bridge_str2prot(optarg, &brprot);
7819                        if (status != DLADM_STATUS_OK)
7820                                die_dlerr(status, "protection %s", optarg);
7821                        break;
7822                case 'R':
7823                        altroot = optarg;
7824                        break;
7825                case 'd':
7826                        if (func == brAdd)
7827                                die_opterr(optopt, option, use);
7828                        if (cfg.field_mask & BR_CFG_DELAY)
7829                                die("forwarding delay set more than once");
7830                        if (!str2int(optarg, &cfg.forward_delay) ||
7831                            cfg.forward_delay < MIN_BR_FWDELAY ||
7832                            cfg.forward_delay > MAX_BR_FWDELAY)
7833                                die("incorrect forwarding delay");
7834                        cfg.field_mask |= BR_CFG_DELAY;
7835                        break;
7836                case 'f':
7837                        if (func == brAdd)
7838                                die_opterr(optopt, option, use);
7839                        if (cfg.field_mask & BR_CFG_FORCE_VER)
7840                                die("force protocol set more than once");
7841                        if (!str2int(optarg, &cfg.force_version) ||
7842                            cfg.force_version < 0)
7843                                die("incorrect force protocol");
7844                        cfg.field_mask |= BR_CFG_FORCE_VER;
7845                        break;
7846                case 'h':
7847                        if (func == brAdd)
7848                                die_opterr(optopt, option, use);
7849                        if (cfg.field_mask & BR_CFG_HELLO)
7850                                die("hello time set more than once");
7851                        if (!str2int(optarg, &cfg.hello_time) ||
7852                            cfg.hello_time < MIN_BR_HELLOT ||
7853                            cfg.hello_time > MAX_BR_HELLOT)
7854                                die("incorrect hello time");
7855                        cfg.field_mask |= BR_CFG_HELLO;
7856                        break;
7857                case 'l':
7858                        if (func == brModify)
7859                                die_opterr(optopt, option, use);
7860                        if (nlink >= MAXPORT)
```



```
7861                                die("too many links specified");
7862                        links[nlink++] = optarg;
7863                        break;
7864                case 'm':
7865                        if (func == brAdd)
7866                                die_opterr(optopt, option, use);
7867                        if (cfg.field_mask & BR_CFG_AGE)
7868                                die("max age set more than once");
7869                        if (!str2int(optarg, &cfg.max_age) ||
7870                            cfg.max_age < MIN_BR_MAXAGE ||
7871                            cfg.max_age > MAX_BR_MAXAGE)
7872                                die("incorrect max age");
7873                        cfg.field_mask |= BR_CFG_AGE;
7874                        break;
7875                case 'p':
7876                        if (func == brAdd)
7877                                die_opterr(optopt, option, use);
7878                        if (cfg.field_mask & BR_CFG_PRIO)
7879                                die("priority set more than once");
7880                        if (!str2int(optarg, &cfg.bridge_priority) ||
7881                            cfg.bridge_priority < MIN_BR_PRIO ||
7882                            cfg.bridge_priority > MAX_BR_PRIO)
7883                                die("incorrect priority");
7884                        cfg.bridge_priority &= 0xF000;
7885                        cfg.field_mask |= BR_CFG_PRIO;
7886                        break;
7887                default:
7888                        die_opterr(optopt, option, use);
7889                        break;
7890                }
7891        }

7893        /* get the bridge name (required last argument) */
7894        if (optind != (argc-1))
7895                usage();

7897        bridge = argv[optind];
7898        if (!dladm_valid_bridgename(bridge))
7899                die("invalid bridge name '%s'", bridge);

7901        /*
7902         * Get the current properties, if any, and merge in with changes.  This
7903         * is necessary (even with the field_mask feature) so that the
7904         * value-checking macros will produce the right results with proposed
7905         * changes to existing configuration.  We only need it for those
7906         * parameters, though.
7907         */
7908        (void) dladm_bridge_get_properties(bridge, &cfg_old, &brprot_old);
7909        if (brprot == DLADM_BRIDGE_PROT_UNKNOWN)
7910                brprot = brprot_old;
7911        if (!(cfg.field_mask & BR_CFG_AGE))
7912                cfg.max_age = cfg_old.max_age;
7913        if (!(cfg.field_mask & BR_CFG_HELLO))
7914                cfg.hello_time = cfg_old.hello_time;
7915        if (!(cfg.field_mask & BR_CFG_DELAY))
7916                cfg.forward_delay = cfg_old.forward_delay;

7918        if (!CHECK_BRIDGE_CONFIG(cfg)) {
7919                warn("illegal forward delay / max age / hello time "
7920                    "combination");
7921                if (NO_MAXAGE(cfg)) {
7922                        die("no max age possible: need forward delay >= %d or "
7923                            "hello time <= %d", MIN_FWDELAY_NOM(cfg),
7924                            MAX_HELLOTIME_NOM(cfg));
7925                } else if (SMALL_MAXAGE(cfg)) {
7926                        if (CAPPED_MAXAGE(cfg))
```

```
7927                                die("max age too small: need age >= %d and "
7928                                    "<= %d or hello time <= %d",
7929                                    MIN_MAXAGE(cfg), MAX_MAXAGE(cfg),
7930                                    MAX_HELLOTIME(cfg));
7931                        else
7932                                die("max age too small: need age >= %d or "
7933                                    "hello time <= %d",
7934                                    MIN_MAXAGE(cfg), MAX_HELLOTIME(cfg));
7935                } else if (FLOORED_MAXAGE(cfg)) {
7936                        die("max age too large: need age >= %d and <= %d or "
7937                            "forward delay >= %d",
7938                            MIN_MAXAGE(cfg), MAX_MAXAGE(cfg),
7939                            MIN_FWDELAY(cfg));
7940                } else {
7941                        die("max age too large: need age <= %d or forward "
7942                            "delay >= %d",
7943                            MAX_MAXAGE(cfg), MIN_FWDELAY(cfg));
7944                }
7945        }

7947        if (altroot != NULL)
7948                altroot_cmd(altroot, argc, argv);

7950        for (n = 0; n < nlink; n++) {
7951                datalink_class_t class;
7952                uint32_t media;
7953                char pointless[DLADM_STRSIZE];

7955                if (dladm_name2info(handle, links[n], &linkids[n], NULL, &class,
7956                    &media) != DLADM_STATUS_OK)
7957                        die("invalid link name '%s'", links[n]);
7958                if (class & ~(DATALINK_CLASS_PHYS | DATALINK_CLASS_AGGR |
7959                    DATALINK_CLASS_ETHERSTUB | DATALINK_CLASS_SIMNET))
7960                        die("%s %s cannot be bridged",
7961                            dladm_class2str(class, pointless), links[n]);
7962                if (media != DL_ETHER && media != DL_100VG &&
7963                    media != DL_ETH_CSMA && media != DL_100BT)
7964                        die("%s interface %s cannot be bridged",
7965                            dladm_media2str(media, pointless), links[n]);
7966        }

7968        if (func == brCreate)
7969                flags |= DLADM_OPT_CREATE;

7971        if (func != brAdd) {
7972                status = dladm_bridge_configure(handle, bridge, &cfg, brprot,
7973                    flags);
7974                if (status != DLADM_STATUS_OK)
7975                        die_dlerr(status, "create operation failed");
7976        }

7978        status = DLADM_STATUS_OK;
7979        for (n = 0; n < nlink; n++) {
7980                status = dladm_bridge_setlink(handle, linkids[n], bridge);
7981                if (status != DLADM_STATUS_OK)
7982                        break;
7983        }

7985        if (n >= nlink) {
7986                /*
7987                 * We were successful.  If we're creating a new bridge, then
7988                 * there's just one more step: enabling.  If we're modifying or
7989                 * just adding links, then we're done.
7990                 */
7991                if (func != brCreate ||
7992                    (status = dladm_bridge_enable(bridge)) == DLADM_STATUS_OK)
```



```
7993                        return;
7994        }

7996        /* clean up the partial configuration */
7997        for (i = 0; i < n; i++)
7998                (void) dladm_bridge_setlink(handle, linkids[i], "");

8000        /* if failure for brCreate, then delete the bridge */
8001        if (func == brCreate)
8002                (void) dladm_bridge_delete(handle, bridge, flags);

8004        if (n < nlink)
8005                die_dlerr(status, "unable to add link %s to bridge %s",
8006                    links[n], bridge);
8007        else
8008                die_dlerr(status, "unable to enable bridge %s", bridge);
8009 }

8011 static void
8012 do_create_bridge(int argc, char **argv, const char *use)
8013 {
8014        create_modify_add_bridge(argc, argv, use, brCreate);
8015 }

8017 static void
8018 do_modify_bridge(int argc, char **argv, const char *use)
8019 {
8020        create_modify_add_bridge(argc, argv, use, brModify);
8021 }

8023 static void
8024 do_add_bridge(int argc, char **argv, const char *use)
8025 {
8026        create_modify_add_bridge(argc, argv, use, brAdd);
8027 }

8029 static void
8030 do_delete_bridge(int argc, char **argv, const char *use)
8031 {
8032        char                    option;
8033        char                    *altroot = NULL;
8034        uint32_t                flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
8035        dladm_status_t          status;

8037        opterr = 0;
8038        while ((option = getopt_long(argc, argv, ":R:", bridge_lopts, NULL)) !=
8039            -1) {
8040                switch (option) {
8041                case 'R':
8042                        altroot = optarg;
8043                        break;
8044                default:
8045                        die_opterr(optopt, option, use);
8046                        break;
8047                }
8048        }

8050        /* get the bridge name (required last argument) */
8051        if (optind != (argc-1))
8052                usage();

8054        if (altroot != NULL)
8055                altroot_cmd(altroot, argc, argv);

8057        status = dladm_bridge_delete(handle, argv[optind], flags);
8058        if (status != DLADM_STATUS_OK)
```

```
8059                 die_dlerr(status, "delete operation failed");
8060 }

8062 static void
8063 do_remove_bridge(int argc, char **argv, const char *use)
8064 {
8065         char            option;
8066         uint_t          n, nlink;
8067         char            *links[MAXPORT];
8068         datalink_id_t   linkids[MAXPORT];
8069         char            *altroot = NULL;
8070         dladm_status_t  status;
8071         boolean_t       removed_one;

8073         nlink = opterr = 0;
8074         while ((option = getopt_long(argc, argv, ":R:l:", bridge_lopts,
8075             NULL)) != -1) {
8076                 switch (option) {
8077                 case 'R':
8078                         altroot = optarg;
8079                         break;
8080                 case 'l':
8081                         if (nlink >= MAXPORT)
8082                                 die("too many links specified");
8083                         links[nlink++] = optarg;
8084                         break;
8085                 default:
8086                         die_opterr(optopt, option, use);
8087                         break;
8088                 }
8089         }

8091         if (nlink == 0)
8092                 usage();

8094         /* get the bridge name (required last argument) */
8095         if (optind != (argc-1))
8096                 usage();

8098         if (altroot != NULL)
8099                 altroot_cmd(altroot, argc, argv);

8101         for (n = 0; n < nlink; n++) {
8102                 char bridge[MAXLINKNAMELEN];

8104                 if (dladm_name2info(handle, links[n], &linkids[n], NULL, NULL,
8105                     NULL) != DLADM_STATUS_OK)
8106                         die("invalid link name '%s'", links[n]);
8107                 status = dladm_bridge_getlink(handle, linkids[n], bridge,
8108                     sizeof (bridge));
8109                 if (status != DLADM_STATUS_OK &&
8110                     status != DLADM_STATUS_NOTFOUND) {
8111                         die_dlerr(status, "cannot get bridge status on %s",
8112                             links[n]);
8113                 }
8114                 if (status == DLADM_STATUS_NOTFOUND ||
8115                     strcmp(bridge, argv[optind]) != 0)
8116                         die("link %s is not on bridge %s", links[n],
8117                             argv[optind]);
8118         }

8120         removed_one = B_FALSE;
8121         for (n = 0; n < nlink; n++) {
8122                 status = dladm_bridge_setlink(handle, linkids[n], "");
8123                 if (status == DLADM_STATUS_OK) {
8124                         removed_one = B_TRUE;
```



```
8125                 } else {
8126                         warn_dlerr(status,
8127                             "cannot remove link %s from bridge %s",
8128                             links[n], argv[optind]);
8129                 }
8130         }
8131         if (!removed_one)
8132                 die("unable to remove any links from bridge %s", argv[optind]);
8133 }

8135 static void
8136 fmt_int(char *buf, size_t buflen, int value, int runvalue,
8137     boolean_t printstar)
8138 {
8139         (void) snprintf(buf, buflen, "%d", value);
8140         if (value != runvalue && printstar)
8141                 (void) strlcat(buf, "*", buflen);
8142 }

8144 static void
8145 fmt_bridge_id(char *buf, size_t buflen, UID_BRIDGE_ID_T *bid)
8146 {
8147         (void) snprintf(buf, buflen, "%u/%x:%x:%x:%x:%x:%x", bid->prio,
8148             bid->addr[0], bid->addr[1], bid->addr[2], bid->addr[3],
8149             bid->addr[4], bid->addr[5]);
8150 }

8152 static dladm_status_t
8153 print_bridge(show_state_t *state, datalink_id_t linkid,
8154     bridge_fields_buf_t *bbuf)
8155 {
8156         char                    link[MAXLINKNAMELEN];
8157         datalink_class_t        class;
8158         uint32_t                flags;
8159         dladm_status_t          status;
8160         UID_STP_CFG_T           smfcfg, runcfg;
8161         UID_STP_STATE_T         stpstate;
8162         dladm_bridge_prot_t     smfprot, runprot;

8164         if ((status = dladm_datalink_id2info(handle, linkid, &flags, &class,
8165             NULL, link, sizeof (link))) != DLADM_STATUS_OK)
8166                 return (status);

8168         if (!(state->ls_flags & flags))
8169                 return (DLADM_STATUS_NOTFOUND);

8171         /* Convert observability node name back to bridge name */
8172         if (!dladm_observe_to_bridge(link))
8173                 return (DLADM_STATUS_NOTFOUND);
8174         (void) strlcpy(bbuf->bridge_name, link, sizeof (bbuf->bridge_name));

8176         /*
8177          * If the running value differs from the one in SMF, and parsable
8178          * output is not requested, then we show the running value with an
8179          * asterisk.
8180          */
8181         (void) dladm_bridge_get_properties(bbuf->bridge_name, &smfcfg,
8182             &smfprot);
8183         (void) dladm_bridge_run_properties(bbuf->bridge_name, &runcfg,
8184             &runprot);
8185         (void) snprintf(bbuf->bridge_protect, sizeof (bbuf->bridge_protect),
8186             "%s%s", state->ls_parsable || smfprot == runprot ? "" : "*",
8187             dladm_bridge_prot2str(runprot));
8188         fmt_int(bbuf->bridge_priority, sizeof (bbuf->bridge_priority),
8189             smfcfg.bridge_priority, runcfg.bridge_priority,
8190             !state->ls_parsable && (runcfg.field_mask & BR_CFG_AGE));
```

```
8191         fmt_int(bbuf->bridge_bmaxage, sizeof (bbuf->bridge_bmaxage),
8192             smfcfg.max_age, runcfg.max_age,
8193             !state->ls_parsable && (runcfg.field_mask & BR_CFG_AGE));
8194         fmt_int(bbuf->bridge_bhellotime,
8195             sizeof (bbuf->bridge_bhellotime), smfcfg.hello_time,
8196             runcfg.hello_time,
8197             !state->ls_parsable && (runcfg.field_mask & BR_CFG_HELLO));
8198         fmt_int(bbuf->bridge_bfwddelay, sizeof (bbuf->bridge_bfwddelay),
8199             smfcfg.forward_delay, runcfg.forward_delay,
8200             !state->ls_parsable && (runcfg.field_mask & BR_CFG_DELAY));
8201         fmt_int(bbuf->bridge_forceproto, sizeof (bbuf->bridge_forceproto),
8202             smfcfg.force_version, runcfg.force_version,
8203             !state->ls_parsable && (runcfg.field_mask & BR_CFG_FORCE_VER));
8204         fmt_int(bbuf->bridge_holdtime, sizeof (bbuf->bridge_holdtime),
8205             smfcfg.hold_time, runcfg.hold_time,
8206             !state->ls_parsable && (runcfg.field_mask & BR_CFG_HOLD_TIME));

8208         if (dladm_bridge_state(bbuf->bridge_name, &stpstate) ==
8209             DLADM_STATUS_OK) {
8210                 fmt_bridge_id(bbuf->bridge_address,
8211                     sizeof (bbuf->bridge_address), &stpstate.bridge_id);
8212                 (void) snprintf(bbuf->bridge_tctime,
8213                     sizeof (bbuf->bridge_tctime), "%lu",
8214                     stpstate.timeSince_Topo_Change);
8215                 (void) snprintf(bbuf->bridge_tccount,
8216                     sizeof (bbuf->bridge_tccount), "%lu",
8217                     stpstate.Topo_Change_Count);
8218                 (void) snprintf(bbuf->bridge_tchange,
8219                     sizeof (bbuf->bridge_tchange), "%u", stpstate.Topo_Change);
8220                 fmt_bridge_id(bbuf->bridge_desroot,
8221                     sizeof (bbuf->bridge_desroot), &stpstate.designated_root);
8222                 (void) snprintf(bbuf->bridge_rootcost,
8223                     sizeof (bbuf->bridge_rootcost), "%lu",
8224                     stpstate.root_path_cost);
8225                 (void) snprintf(bbuf->bridge_rootport,
8226                     sizeof (bbuf->bridge_rootport), "%u", stpstate.root_port);
8227                 (void) snprintf(bbuf->bridge_maxage,
8228                     sizeof (bbuf->bridge_maxage), "%d", stpstate.max_age);
8229                 (void) snprintf(bbuf->bridge_hellotime,
8230                     sizeof (bbuf->bridge_hellotime), "%d", stpstate.hello_time);
8231                 (void) snprintf(bbuf->bridge_fwddelay,
8232                     sizeof (bbuf->bridge_fwddelay), "%d",
8233                     stpstate.forward_delay);
8234         }
8235         return (DLADM_STATUS_OK);
8236 }

8238 static dladm_status_t
8239 print_bridge_stats(show_state_t *state, datalink_id_t linkid,
8240     bridge_statfields_buf_t *bbuf)
8241 {
8242         char                    link[MAXLINKNAMELEN];
8243         datalink_class_t        class;
8244         uint32_t                flags;
8245         dladm_status_t          status;
8246         kstat_ctl_t             *kcp;
8247         kstat_t                 *ksp;
8248         brsum_t                 *brsum = (brsum_t *)&state->ls_prevstats;
8249         brsum_t                 newval;

8251 #ifndef lint
8252         /* This is a compile-time assertion; optimizer normally fixes this */
8253         extern void brsum_t_is_too_large(void);

8255         if (sizeof (*brsum) > sizeof (state->ls_prevstats))
8256                 brsum_t_is_too_large();
```



```
8257 #endif

8259         if (state->ls_firstonly) {
8260                 if (state->ls_donefirst)
8261                         return (DLADM_WALK_CONTINUE);
8262                 state->ls_donefirst = B_TRUE;
8263         } else {
8264                 bzero(brsum, sizeof (*brsum));
8265         }
8266         bzero(&newval, sizeof (newval));

8268         if ((status = dladm_datalink_id2info(handle, linkid, &flags, &class,
8269             NULL, link, sizeof (link))) != DLADM_STATUS_OK)
8270                 return (status);

8272         if (!(state->ls_flags & flags))
8273                 return (DLADM_STATUS_NOTFOUND);

8275         if ((kcp = kstat_open()) == NULL) {
8276                 warn("kstat open operation failed");
8277                 return (DLADM_STATUS_OK);
8278         }
8279         if ((ksp = kstat_lookup(kcp, "bridge", 0, link)) != NULL &&
8280             kstat_read(kcp, ksp, NULL) != -1) {
8281                 if (dladm_kstat_value(ksp, "drops", KSTAT_DATA_UINT64,
8282                     &newval.drops) == DLADM_STATUS_OK) {
8283                         (void) snprintf(bbuf->bridges_drops,
8284                             sizeof (bbuf->bridges_drops), "%llu",
8285                             newval.drops - brsum->drops);
8286                 }
8287                 if (dladm_kstat_value(ksp, "forward_direct", KSTAT_DATA_UINT64,
8288                     &newval.forward_dir) == DLADM_STATUS_OK) {
8289                         (void) snprintf(bbuf->bridges_forwards,
8290                             sizeof (bbuf->bridges_forwards), "%llu",
8291                             newval.forward_dir - brsum->forward_dir);
8292                 }
8293                 if (dladm_kstat_value(ksp, "forward_mbcast", KSTAT_DATA_UINT64,
8294                     &newval.forward_mb) == DLADM_STATUS_OK) {
8295                         (void) snprintf(bbuf->bridges_mbcast,
8296                             sizeof (bbuf->bridges_mbcast), "%llu",
8297                             newval.forward_mb - brsum->forward_mb);
8298                 }
8299                 if (dladm_kstat_value(ksp, "forward_unknown", KSTAT_DATA_UINT64,
8300                     &newval.forward_unk) == DLADM_STATUS_OK) {
8301                         (void) snprintf(bbuf->bridges_unknown,
8302                             sizeof (bbuf->bridges_unknown), "%llu",
8303                             newval.forward_unk - brsum->forward_unk);
8304                 }
8305                 if (dladm_kstat_value(ksp, "recv", KSTAT_DATA_UINT64,
8306                     &newval.recv) == DLADM_STATUS_OK) {
8307                         (void) snprintf(bbuf->bridges_recv,
8308                             sizeof (bbuf->bridges_recv), "%llu",
8309                             newval.recv - brsum->recv);
8310                 }
8311                 if (dladm_kstat_value(ksp, "sent", KSTAT_DATA_UINT64,
8312                     &newval.sent) == DLADM_STATUS_OK) {
8313                         (void) snprintf(bbuf->bridges_sent,
8314                             sizeof (bbuf->bridges_sent), "%llu",
8315                             newval.sent - brsum->sent);
8316                 }
8317         }
8318         (void) kstat_close(kcp);

8320         /* Convert observability node name back to bridge name */
8321         if (!dladm_observe_to_bridge(link))
8322                 return (DLADM_STATUS_NOTFOUND);
```

```
8323         (void) strlcpy(bbuf->bridges_name, link, sizeof (bbuf->bridges_name));

8325         *brsum = newval;

8327         return (DLADM_STATUS_OK);
8328 }

8330 /*
8331  * This structure carries around extra state information for the show-bridge
8332  * command and allows us to use common support functions.
8333  */
8334 typedef struct {
8335         show_state_t    state;
8336         boolean_t       show_stats;
8337         const char      *bridge;
8338 } show_brstate_t;

8340 /* ARGSUSED */
8341 static int
8342 show_bridge(dladm_handle_t handle, datalink_id_t linkid, void *arg)
8343 {
8344         show_brstate_t  *brstate = arg;
8345         void *buf;

8347         if (brstate->show_stats) {
8348                 bridge_statfields_buf_t bbuf;

8350                 bzero(&bbuf, sizeof (bbuf));
8351                 brstate->state.ls_status = print_bridge_stats(&brstate->state,
8352                     linkid, &bbuf);
8353                 buf = &bbuf;
8354         } else {
8355                 bridge_fields_buf_t bbuf;

8357                 bzero(&bbuf, sizeof (bbuf));
8358                 brstate->state.ls_status = print_bridge(&brstate->state, linkid,
8359                     &bbuf);
8360                 buf = &bbuf;
8361         }
8362         if (brstate->state.ls_status == DLADM_STATUS_OK)
8363                 ofmt_print(brstate->state.ls_ofmt, buf);
8364         return (DLADM_WALK_CONTINUE);
8365 }

8367 static void
8368 fmt_bool(char *buf, size_t buflen, int val)
8369 {
8370         (void) strlcpy(buf, val ? "yes" : "no", buflen);
8371 }

8373 static dladm_status_t
8374 print_bridge_link(show_state_t *state, datalink_id_t linkid,
8375     bridge_link_fields_buf_t *bbuf)
8376 {
8377         datalink_class_t        class;
8378         uint32_t                flags;
8379         dladm_status_t          status;
8380         UID_STP_PORT_STATE_T    stpstate;

8382         status = dladm_datalink_id2info(handle, linkid, &flags, &class, NULL,
8383             bbuf->bridgel_link, sizeof (bbuf->bridgel_link));
8384         if (status != DLADM_STATUS_OK)
8385                 return (status);

8387         if (!(state->ls_flags & flags))
8388                 return (DLADM_STATUS_NOTFOUND);
```



```
8390         if (dladm_bridge_link_state(handle, linkid, &stpstate) ==
8391             DLADM_STATUS_OK) {
8392                 (void) snprintf(bbuf->bridgel_index,
8393                     sizeof (bbuf->bridgel_index), "%u", stpstate.port_no);
8394                 if (dlsym(RTLD_PROBE, "STP_IN_state2str")) {
8395                         (void) strlcpy(bbuf->bridgel_state,
8396                             STP_IN_state2str(stpstate.state),
8397                             sizeof (bbuf->bridgel_state));
8398                 } else {
8399                         (void) snprintf(bbuf->bridgel_state,
8400                             sizeof (bbuf->bridgel_state), "%u",
8401                             stpstate.state);
8402                 }
8403                 (void) snprintf(bbuf->bridgel_uptime,
8404                     sizeof (bbuf->bridgel_uptime), "%lu", stpstate.uptime);
8405                 (void) snprintf(bbuf->bridgel_opercost,
8406                     sizeof (bbuf->bridgel_opercost), "%lu",
8407                     stpstate.oper_port_path_cost);
8408                 fmt_bool(bbuf->bridgel_operp2p, sizeof (bbuf->bridgel_operp2p),
8409                     stpstate.oper_point2point);
8410                 fmt_bool(bbuf->bridgel_operedge,
8411                     sizeof (bbuf->bridgel_operedge), stpstate.oper_edge);
8412                 fmt_bridge_id(bbuf->bridgel_desroot,
8413                     sizeof (bbuf->bridgel_desroot), &stpstate.designated_root);
8414                 (void) snprintf(bbuf->bridgel_descost,
8415                     sizeof (bbuf->bridgel_descost), "%lu",
8416                     stpstate.designated_cost);
8417                 fmt_bridge_id(bbuf->bridgel_desbridge,
8418                     sizeof (bbuf->bridgel_desbridge),
8419                     &stpstate.designated_bridge);
8420                 (void) snprintf(bbuf->bridgel_desport,
8421                     sizeof (bbuf->bridgel_desport), "%u",
8422                     stpstate.designated_port);
8423                 fmt_bool(bbuf->bridgel_tcack, sizeof (bbuf->bridgel_tcack),
8424                     stpstate.top_change_ack);
8425         }
8426         return (DLADM_STATUS_OK);
8427 }

8429 static dladm_status_t
8430 print_bridge_link_stats(show_state_t *state, datalink_id_t linkid,
8431     bridge_link_statfields_buf_t *bbuf)
8432 {
8433         datalink_class_t        class;
8434         uint32_t                flags;
8435         dladm_status_t          status;
8436         UID_STP_PORT_STATE_T    stpstate;
8437         kstat_ctl_t             *kcp;
8438         kstat_t                 *ksp;
8439         char                    bridge[MAXLINKNAMELEN];
8440         char                    kstatname[MAXLINKNAMELEN*2 + 1];
8441         brlsum_t                *brlsum = (brlsum_t *)&state->ls_prevstats;
8442         brlsum_t                newval;

8444 #ifndef lint
8445         /* This is a compile-time assertion; optimizer normally fixes this */
8446         extern void brlsum_t_is_too_large(void);

8448         if (sizeof (*brlsum) > sizeof (state->ls_prevstats))
8449                 brlsum_t_is_too_large();
8450 #endif

8452         if (state->ls_firstonly) {
8453                 if (state->ls_donefirst)
8454                         return (DLADM_WALK_CONTINUE);
```

```
8455                 state->ls_donefirst = B_TRUE;
8456         } else {
8457                 bzero(brlsum, sizeof (*brlsum));
8458         }
8459         bzero(&newval, sizeof (newval));

8461         status = dladm_datalink_id2info(handle, linkid, &flags, &class, NULL,
8462             bbuf->bridgels_link, sizeof (bbuf->bridgels_link));
8463         if (status != DLADM_STATUS_OK)
8464                 return (status);

8466         if (!(state->ls_flags & flags))
8467                 return (DLADM_STATUS_NOTFOUND);

8469         if (dladm_bridge_link_state(handle, linkid, &stpstate) ==
8470             DLADM_STATUS_OK) {
8471                 newval.cfgbpdu = stpstate.rx_cfg_bpdu_cnt;
8472                 newval.tcnbpdu = stpstate.rx_tcn_bpdu_cnt;
8473                 newval.rstpbpdu = stpstate.rx_rstp_bpdu_cnt;
8474                 newval.txbpdu = stpstate.txCount;

8476                 (void) snprintf(bbuf->bridgels_cfgbpdu,
8477                     sizeof (bbuf->bridgels_cfgbpdu), "%lu",
8478                     newval.cfgbpdu - brlsum->cfgbpdu);
8479                 (void) snprintf(bbuf->bridgels_tcnbpdu,
8480                     sizeof (bbuf->bridgels_tcnbpdu), "%lu",
8481                     newval.tcnbpdu - brlsum->tcnbpdu);
8482                 (void) snprintf(bbuf->bridgels_rstpbpdu,
8483                     sizeof (bbuf->bridgels_rstpbpdu), "%lu",
8484                     newval.rstpbpdu - brlsum->rstpbpdu);
8485                 (void) snprintf(bbuf->bridgels_txbpdu,
8486                     sizeof (bbuf->bridgels_txbpdu), "%lu",
8487                     newval.txbpdu - brlsum->txbpdu);
8488         }

8490         if ((status = dladm_bridge_getlink(handle, linkid, bridge,
8491             sizeof (bridge))) != DLADM_STATUS_OK)
8492                 goto bls_out;
8493         (void) snprintf(kstatname, sizeof (kstatname), "%s0-%s", bridge,
8494             bbuf->bridgels_link);
8495         if ((kcp = kstat_open()) == NULL) {
8496                 warn("kstat open operation failed");
8497                 goto bls_out;
8498         }
8499         if ((ksp = kstat_lookup(kcp, "bridge", 0, kstatname)) != NULL &&
8500             kstat_read(kcp, ksp, NULL) != -1) {
8501                 if (dladm_kstat_value(ksp, "drops", KSTAT_DATA_UINT64,
8502                     &newval.drops) != -1) {
8503                         (void) snprintf(bbuf->bridgels_drops,
8504                             sizeof (bbuf->bridgels_drops), "%llu",
8505                             newval.drops - brlsum->drops);
8506                 }
8507                 if (dladm_kstat_value(ksp, "recv", KSTAT_DATA_UINT64,
8508                     &newval.recv) != -1) {
8509                         (void) snprintf(bbuf->bridgels_recv,
8510                             sizeof (bbuf->bridgels_recv), "%llu",
8511                             newval.recv - brlsum->recv);
8512                 }
8513                 if (dladm_kstat_value(ksp, "xmit", KSTAT_DATA_UINT64,
8514                     &newval.xmit) != -1) {
8515                         (void) snprintf(bbuf->bridgels_xmit,
8516                             sizeof (bbuf->bridgels_xmit), "%llu",
8517                             newval.xmit - brlsum->xmit);
8518                 }
8519         }
8520         (void) kstat_close(kcp);
```



```
8521 bls_out:
8522         *brlsum = newval;

8524         return (status);
8525 }

8527 static void
8528 show_bridge_link(datalink_id_t linkid, show_brstate_t *brstate)
8529 {
8530         void *buf;

8532         if (brstate->show_stats) {
8533                 bridge_link_statfields_buf_t bbuf;

8535                 bzero(&bbuf, sizeof (bbuf));
8536                 brstate->state.ls_status = print_bridge_link_stats(
8537                     &brstate->state, linkid, &bbuf);
8538                 buf = &bbuf;
8539         } else {
8540                 bridge_link_fields_buf_t bbuf;

8542                 bzero(&bbuf, sizeof (bbuf));
8543                 brstate->state.ls_status = print_bridge_link(&brstate->state,
8544                     linkid, &bbuf);
8545                 buf = &bbuf;
8546         }
8547         if (brstate->state.ls_status == DLADM_STATUS_OK)
8548                 ofmt_print(brstate->state.ls_ofmt, buf);
8549 }

8551 /* ARGSUSED */
8552 static int
8553 show_bridge_link_walk(dladm_handle_t handle, datalink_id_t linkid, void *arg)
8554 {
8555         show_brstate_t  *brstate = arg;
8556         char bridge[MAXLINKNAMELEN];

8558         if (dladm_bridge_getlink(handle, linkid, bridge, sizeof (bridge)) ==
8559             DLADM_STATUS_OK && strcmp(bridge, brstate->bridge) == 0) {
8560                 show_bridge_link(linkid, brstate);
8561         }
8562         return (DLADM_WALK_CONTINUE);
8563 }

8565 static void
8566 show_bridge_fwd(dladm_handle_t handle, bridge_listfwd_t *blf,
8567     show_state_t *state)
8568 {
8569         bridge_fwd_fields_buf_t bbuf;

8571         bzero(&bbuf, sizeof (bbuf));
8572         (void) snprintf(bbuf.bridgef_dest, sizeof (bbuf.bridgef_dest),
8573             "%s", ether_ntoa((struct ether_addr *)blf->blf_dest));
8574         if (blf->blf_is_local) {
8575                 (void) strlcpy(bbuf.bridgef_flags, "L",
8576                     sizeof (bbuf.bridgef_flags));
8577         } else {
8578                 (void) snprintf(bbuf.bridgef_age, sizeof (bbuf.bridgef_age),
8579                     "%2d.%03d", blf->blf_ms_age / 1000, blf->blf_ms_age % 1000);
8580                 if (blf->blf_trill_nick != 0) {
8581                         (void) snprintf(bbuf.bridgef_output,
8582                             sizeof (bbuf.bridgef_output), "%u",
8583                             blf->blf_trill_nick);
8584                 }
8585         }
8586         if (blf->blf_linkid != DATALINK_INVALID_LINKID &&
```

```
8587             blf->blf_trill_nick == 0) {
8588                 state->ls_status = dladm_datalink_id2info(handle,
8589                     blf->blf_linkid, NULL, NULL, NULL, bbuf.bridgef_output,
8590                     sizeof (bbuf.bridgef_output));
8591         }
8592         if (state->ls_status == DLADM_STATUS_OK)
8593                 ofmt_print(state->ls_ofmt, &bbuf);
8594 }

8596 static void
8597 show_bridge_trillnick(trill_listnick_t *tln, show_state_t *state)
8598 {
8599         bridge_trill_fields_buf_t bbuf;

8601         bzero(&bbuf, sizeof (bbuf));
8602         (void) snprintf(bbuf.bridget_nick, sizeof (bbuf.bridget_nick),
8603             "%u", tln->tln_nick);
8604         if (tln->tln_ours) {
8605                 (void) strlcpy(bbuf.bridget_flags, "L",
8606                     sizeof (bbuf.bridget_flags));
8607         } else {
8608                 state->ls_status = dladm_datalink_id2info(handle,
8609                     tln->tln_linkid, NULL, NULL, NULL, bbuf.bridget_link,
8610                     sizeof (bbuf.bridget_link));
8611                 (void) snprintf(bbuf.bridget_nexthop,
8612                     sizeof (bbuf.bridget_nexthop), "%s",
8613                     ether_ntoa((struct ether_addr *)tln->tln_nexthop));
8614         }
8615         if (state->ls_status == DLADM_STATUS_OK)
8616                 ofmt_print(state->ls_ofmt, &bbuf);
8617 }

8619 static void
8620 do_show_bridge(int argc, char **argv, const char *use)
8621 {
8622         int             option;
8623         enum {
8624                 bridgeMode, linkMode, fwdMode, trillMode
8625         }               op_mode = bridgeMode;
8626         uint32_t        flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
8627         boolean_t       parsable = B_FALSE;
8628         datalink_id_t   linkid = DATALINK_ALL_LINKID;
8629         int             interval = 0;
8630         show_brstate_t  brstate;
8631         dladm_status_t  status;
8632         char            *fields_str = NULL;
8633         /* default: bridge-related data */
8634         char            *all_fields = "bridge,protect,address,priority,bmaxage,"
8635             "bhellotime,bfwddelay,forceproto,tctime,tccount,tchange,"
8636             "desroot,rootcost,rootport,maxage,hellotime,fwddelay,holdtime";
8637         char            *default_fields = "bridge,protect,address,priority,"
8638             "desroot";
8639         char            *all_statfields = "bridge,drops,forwards,mbcast,"
8640             "unknown,recv,sent";
8641         char            *default_statfields = "bridge,drops,forwards,mbcast,"
8642             "unknown";
8643         /* -l: link-related data */
8644         char            *all_link_fields = "link,index,state,uptime,opercost,"
8645             "operp2p,operedge,desroot,descost,desbridge,desport,tcack";
8646         char            *default_link_fields = "link,state,uptime,desroot";
8647         char            *all_link_statfields = "link,cfgbpdu,tcnbpdu,rstpbpdu,"
8648             "txbpdu,drops,recv,xmit";
8649         char            *default_link_statfields = "link,drops,recv,xmit";
8650         /* -f: bridge forwarding table related data */
8651         char            *default_fwd_fields = "dest,age,flags,output";
8652         /* -t: TRILL nickname table related data */
```



```
8653         char            *default_trill_fields = "nick,flags,link,nexthop";
8654         char            *default_str;
8655         char            *all_str;
8656         ofmt_field_t    *field_arr;
8657         ofmt_handle_t   ofmt;
8658         ofmt_status_t   oferr;
8659         uint_t          ofmtflags = 0;

8661         bzero(&brstate, sizeof (brstate));

8663         opterr = 0;
8664         while ((option = getopt_long(argc, argv, ":fi:lo:pst",
8665             bridge_show_lopts, NULL)) != -1) {
8666                 switch (option) {
8667                 case 'f':
8668                         if (op_mode != bridgeMode && op_mode != fwdMode)
8669                                 die("-f is incompatible with -l or -t");
8670                         op_mode = fwdMode;
8671                         break;
8672                 case 'i':
8673                         if (interval != 0)
8674                                 die_optdup(option);
8675                         if (!str2int(optarg, &interval) || interval == 0)
8676                                 die("invalid interval value '%s'", optarg);
8677                         break;
8678                 case 'l':
8679                         if (op_mode != bridgeMode && op_mode != linkMode)
8680                                 die("-l is incompatible with -f or -t");
8681                         op_mode = linkMode;
8682                         break;
8683                 case 'o':
8684                         fields_str = optarg;
8685                         break;
8686                 case 'p':
8687                         if (parsable)
8688                                 die_optdup(option);
8689                         parsable = B_TRUE;
8690                         break;
8691                 case 's':
8692                         if (brstate.show_stats)
8693                                 die_optdup(option);
8694                         brstate.show_stats = B_TRUE;
8695                         break;
8696                 case 't':
8697                         if (op_mode != bridgeMode && op_mode != trillMode)
8698                                 die("-t is incompatible with -f or -l");
8699                         op_mode = trillMode;
8700                         break;
8701                 default:
8702                         die_opterr(optopt, option, use);
8703                         break;
8704                 }
8705         }

8707         if (interval != 0 && !brstate.show_stats)
8708                 die("the -i option can be used only with -s");

8710         if ((op_mode == fwdMode || op_mode == trillMode) && brstate.show_stats)
8711                 die("the -f/-t and -s options cannot be used together");

8713         /* get the bridge name (optional last argument) */
8714         if (optind == (argc-1)) {
8715                 char lname[MAXLINKNAMELEN];
8716                 uint32_t lnkflg;
8717                 datalink_class_t class;
```

```
8719                brstate.bridge = argv[optind];
8720                (void) snprintf(lname, sizeof (lname), "%s0", brstate.bridge);
8721                if ((status = dladm_name2info(handle, lname, &linkid, &lnkflg,
8722                    &class, NULL)) != DLADM_STATUS_OK) {
8723                        die_dlerr(status, "bridge %s is not valid",
8724                            brstate.bridge);
8725                }

8727                if (class != DATALINK_CLASS_BRIDGE)
8728                        die("%s is not a bridge", brstate.bridge);

8730                if (!(lnkflg & flags)) {
8731                        die_dlerr(DLADM_STATUS_BADARG,
8732                            "bridge %s is temporarily removed", brstate.bridge);
8733                }
8734        } else if (optind != argc) {
8735                usage();
8736        } else if (op_mode != bridgeMode) {
8737                die("bridge name required for -l, -f, or -t");
8738                return;
8739        }

8741        brstate.state.ls_parsable = parsable;
8742        brstate.state.ls_flags = flags;
8743        brstate.state.ls_firstonly = (interval != 0);

8745        switch (op_mode) {
8746        case bridgeMode:
8747                if (brstate.show_stats) {
8748                        default_str = default_statfields;
8749                        all_str = all_statfields;
8750                        field_arr = bridge_statfields;
8751                } else {
8752                        default_str = default_fields;
8753                        all_str = all_fields;
8754                        field_arr = bridge_fields;
8755                }
8756                break;

8758        case linkMode:
8759                if (brstate.show_stats) {
8760                        default_str = default_link_statfields;
8761                        all_str = all_link_statfields;
8762                        field_arr = bridge_link_statfields;
8763                } else {
8764                        default_str = default_link_fields;
8765                        all_str = all_link_fields;
8766                        field_arr = bridge_link_fields;
8767                }
8768                break;

8770        case fwdMode:
8771                default_str = all_str = default_fwd_fields;
8772                field_arr = bridge_fwd_fields;
8773                break;

8775        case trillMode:
8776                default_str = all_str = default_trill_fields;
8777                field_arr = bridge_trill_fields;
8778                break;
8779        }

8781        if (fields_str == NULL)
8782                fields_str = default_str;
8783        else if (strcasecmp(fields_str, "all") == 0)
8784                fields_str = all_str;
```



```
8786        if (parsable)
8787                ofmtflags |= OFMT_PARSABLE;
8788        oferr = ofmt_open(fields_str, field_arr, ofmtflags, 0, &ofmt);
8789        dladm_ofmt_check(oferr, brstate.state.ls_parsable, ofmt);
8790        brstate.state.ls_ofmt = ofmt;

8792        for (;;) {
8793                brstate.state.ls_donefirst = B_FALSE;
8794                switch (op_mode) {
8795                case bridgeMode:
8796                        if (linkid == DATALINK_ALL_LINKID) {
8797                                (void) dladm_walk_datalink_id(show_bridge,
8798                                    handle, &brstate, DATALINK_CLASS_BRIDGE,
8799                                    DATALINK_ANY_MEDIATYPE, flags);
8800                        } else {
8801                                (void) show_bridge(handle, linkid, &brstate);
8802                                if (brstate.state.ls_status !=
8803                                    DLADM_STATUS_OK) {
8804                                        die_dlerr(brstate.state.ls_status,
8805                                            "failed to show bridge %s",
8806                                            brstate.bridge);
8807                                }
8808                        }
8809                        break;

8811                case linkMode: {
8812                        datalink_id_t *dlp;
8813                        uint_t i, nlinks;

8815                        dlp = dladm_bridge_get_portlist(brstate.bridge,
8816                            &nlinks);
8817                        if (dlp != NULL) {
8818                                for (i = 0; i < nlinks; i++)
8819                                        show_bridge_link(dlp[i], &brstate);
8820                                dladm_bridge_free_portlist(dlp);
8821                        } else if (errno == ENOENT) {
8822                                /* bridge not running; iterate on libdladm */
8823                                (void) dladm_walk_datalink_id(
8824                                    show_bridge_link_walk, handle,
8825                                    &brstate, DATALINK_CLASS_PHYS |
8826                                    DATALINK_CLASS_AGGR |
8827                                    DATALINK_CLASS_ETHERSTUB,
8828                                    DATALINK_ANY_MEDIATYPE, flags);
8829                        } else {
8830                                die("unable to get port list for bridge %s: %s",
8831                                    brstate.bridge, strerror(errno));
8832                        }
8833                        break;
8834                }

8836                case fwdMode: {
8837                        bridge_listfwd_t *blf;
8838                        uint_t i, nfwd;

8840                        blf = dladm_bridge_get_fwdtable(handle, brstate.bridge,
8841                            &nfwd);
8842                        if (blf == NULL) {
8843                                die("unable to get forwarding entries for "
8844                                    "bridge %s", brstate.bridge);
8845                        } else {
8846                                for (i = 0; i < nfwd; i++)
8847                                        show_bridge_fwd(handle, blf + i,
8848                                            &brstate.state);
8849                                dladm_bridge_free_fwdtable(blf);
8850                        }
```

```
8851                        break;
8852                }

8854                case trillMode: {
8855                        trill_listnick_t *tln;
8856                        uint_t i, nnick;

8858                        tln = dladm_bridge_get_trillnick(brstate.bridge,
8859                            &nnick);
8860                        if (tln == NULL) {
8861                                if (errno == ENOENT)
8862                                        die("bridge %s is not running TRILL",
8863                                            brstate.bridge);
8864                                else
8865                                        die("unable to get TRILL nickname "
8866                                            "entries for bridge %s",
8867                                            brstate.bridge);
8868                        } else {
8869                                for (i = 0; i < nnick; i++)
8870                                        show_bridge_trillnick(tln + i,
8871                                            &brstate.state);
8872                                dladm_bridge_free_trillnick(tln);
8873                        }
8874                        break;
8875                }
8876                }
8877                if (interval == 0)
8878                        break;
8879                (void) sleep(interval);
8880        }
8881 }

8883 /*
8884  * "-R" option support. It is used for live upgrading. Append dladm commands
8885  * to a upgrade script which will be run when the alternative root boots up:
8886  *
8887  * - If the /etc/dladm/datalink.conf file exists on the alternative root,
8888  * append dladm commands to the <altroot>/var/svc/profile/upgrade_datalink
8889  * script. This script will be run as part of the network/physical service.
8890  * We cannot defer this to /var/svc/profile/upgrade because then the
8891  * configuration will not be able to take effect before network/physical
8892  * plumbs various interfaces.
8893  *
8894  * - If the /etc/dladm/datalink.conf file does not exist on the alternative
8895  * root, append dladm commands to the <altroot>/var/svc/profile/upgrade script,
8896  * which will be run in the manifest-import service.
8897  *
8898  * Note that the SMF team is considering to move the manifest-import service
8899  * to be run at the very begining of boot. Once that is done, the need for
8900  * the /var/svc/profile/upgrade_datalink script will not exist any more.
8901  */
8902 static void
8903 altroot_cmd(char *altroot, int argc, char *argv[])
8904 {
8905         char            path[MAXPATHLEN];
8906         struct stat     stbuf;
8907         FILE            *fp;
8908         int             i;

8910         /*
8911          * Check for the existence of the /etc/dladm/datalink.conf
8912          * configuration file, and determine the name of script file.
8913          */
8914         (void) snprintf(path, MAXPATHLEN, "/%s/etc/dladm/datalink.conf",
8915             altroot);
8916         if (stat(path, &stbuf) < 0) {
```



```
8917                 (void) snprintf(path, MAXPATHLEN, "/%s/%s", altroot,
8918                     SMF_UPGRADE_FILE);
8919         } else {
8920                 (void) snprintf(path, MAXPATHLEN, "/%s/%s", altroot,
8921                     SMF_UPGRADEDATALINK_FILE);
8922         }

8924         if ((fp = fopen(path, "a+")) == NULL)
8925                 die("operation not supported on %s", altroot);

8927         (void) fprintf(fp, "/sbin/dladm ");
8928         for (i = 0; i < argc; i++) {
8929                 /*
8930                  * Directly write to the file if it is not the "-R <altroot>"
8931                  * option. In which case, skip it.
8932                  */
8933                 if (strcmp(argv[i], "-R") != 0)
8934                         (void) fprintf(fp, "%s ", argv[i]);
8935                 else
8936                         i ++;
8937         }
8938         (void) fprintf(fp, "%s\n", SMF_DLADM_UPGRADE_MSG);
8939         (void) fclose(fp);
8940         dladm_close(handle);
8941         exit(EXIT_SUCCESS);
8942 }

8944 /*
8945  * Convert the string to an integer. Note that the string must not have any
8946  * trailing non-integer characters.
8947  */
8948 static boolean_t
8949 str2int(const char *str, int *valp)
8950 {
8951         int     val;
8952         char    *endp = NULL;

8954         errno = 0;
8955         val = strtol(str, &endp, 10);
8956         if (errno != 0 || *endp != '\0')
8957                 return (B_FALSE);

8959         *valp = val;
8960         return (B_TRUE);
8961 }

8963 /* PRINTFLIKE1 */
8964 static void
8965 warn(const char *format, ...)
8966 {
8967         va_list alist;

8969         format = gettext(format);
8970         (void) fprintf(stderr, "%s: warning: ", progname);

8972         va_start(alist, format);
8973         (void) vfprintf(stderr, format, alist);
8974         va_end(alist);

8976         (void) putc('\n', stderr);
8977 }

8979 /* PRINTFLIKE2 */
8980 static void
8981 warn_dlerr(dladm_status_t err, const char *format, ...)
8982 {
```

```
8983         va_list alist;
8984         char    errmsg[DLADM_STRSIZE];

8986         format = gettext(format);
8987         (void) fprintf(stderr, gettext("%s: warning: "), progname);

8989         va_start(alist, format);
8990         (void) vfprintf(stderr, format, alist);
8991         va_end(alist);
8992         (void) fprintf(stderr, ": %s\n", dladm_status2str(err, errmsg));
8993 }

8995 /*
8996  * Also closes the dladm handle if it is not NULL.
8997  */
8998 /* PRINTFLIKE2 */
8999 static void
9000 die_dlerr(dladm_status_t err, const char *format, ...)
9001 {
9002         va_list alist;
9003         char    errmsg[DLADM_STRSIZE];

9005         format = gettext(format);
9006         (void) fprintf(stderr, "%s: ", progname);

9008         va_start(alist, format);
9009         (void) vfprintf(stderr, format, alist);
9010         va_end(alist);
9011         (void) fprintf(stderr, ": %s\n", dladm_status2str(err, errmsg));

9013         /* close dladm handle if it was opened */
9014         if (handle != NULL)
9015                 dladm_close(handle);

9017         exit(EXIT_FAILURE);
9018 }

9020 /* PRINTFLIKE1 */
9021 static void
9022 die(const char *format, ...)
9023 {
9024         va_list alist;

9026         format = gettext(format);
9027         (void) fprintf(stderr, "%s: ", progname);

9029         va_start(alist, format);
9030         (void) vfprintf(stderr, format, alist);
9031         va_end(alist);

9033         (void) putc('\n', stderr);

9035         /* close dladm handle if it was opened */
9036         if (handle != NULL)
9037                 dladm_close(handle);

9039         exit(EXIT_FAILURE);
9040 }

9042 static void
9043 die_optdup(int opt)
9044 {
9045         die("the option -%c cannot be specified more than once", opt);
9046 }

9048 static void
```



```
9049 die_opterr(int opt, int opterr, const char *usage)
9050 {
9051         switch (opterr) {
9052         case ':':
9053                 die("option '-%c' requires a value\nusage: %s", opt,
9054                     gettext(usage));
9055                 break;
9056         case '?':
9057         default:
9058                 die("unrecognized option '-%c'\nusage: %s", opt,
9059                     gettext(usage));
9060                 break;
9061         }
9062 }

9064 static void
9065 show_ether_xprop(void *arg, dladm_ether_info_t *eattr)
9066 {
9067         print_ether_state_t     *statep = arg;
9068         ether_fields_buf_t      ebuf;
9069         int                     i;

9071         for (i = CAPABLE; i <= PEERADV; i++)  {
9072                 bzero(&ebuf, sizeof (ebuf));
9073                 (void) strlcpy(ebuf.eth_ptype, ptype[i],
9074                     sizeof (ebuf.eth_ptype));
9075                 (void) dladm_ether_autoneg2str(ebuf.eth_autoneg,
9076                     sizeof (ebuf.eth_autoneg), eattr, i);
9077                 (void) dladm_ether_spdx2str(ebuf.eth_spdx,
9078                     sizeof (ebuf.eth_spdx), eattr, i);
9079                 (void) dladm_ether_pause2str(ebuf.eth_pause,
9080                     sizeof (ebuf.eth_pause), eattr, i);
9081                 (void) strlcpy(ebuf.eth_rem_fault,
9082                     (eattr->lei_attr[i].le_fault ? "fault" : "none"),
9083                     sizeof (ebuf.eth_rem_fault));
9084                 ofmt_print(statep->es_ofmt, &ebuf);
9085         }

9087 }

9089 static boolean_t
9090 link_is_ether(const char *link, datalink_id_t *linkid)
9091 {
9092         uint32_t media;
9093         datalink_class_t class;

9095         if (dladm_name2info(handle, link, linkid, NULL, &class, &media) ==
9096             DLADM_STATUS_OK) {
9097                 if (class == DATALINK_CLASS_PHYS && media == DL_ETHER)
9098                         return (B_TRUE);
9099         }
9100         return (B_FALSE);
9101 }

9103 /*
9104  * default output callback function that, when invoked,
9105  * prints string which is offset by ofmt_arg->ofmt_id within buf.
9106  */
9107 static boolean_t
9108 print_default_cb(ofmt_arg_t *ofarg, char *buf, uint_t bufsize)
9109 {
9110         char *value;

9112         value = (char *)ofarg->ofmt_cbarg + ofarg->ofmt_id;
9113         (void) strlcpy(buf, value, bufsize);
9114         return (B_TRUE);
```

```
9115 }

9117 static void
9118 dladm_ofmt_check(ofmt_status_t oferr, boolean_t parsable,
9119     ofmt_handle_t ofmt)
9120 {
9121         char buf[OFMT_BUFSIZE];

9123         if (oferr == OFMT_SUCCESS)
9124                 return;
9125         (void) ofmt_strerror(ofmt, oferr, buf, sizeof (buf));
9126         /*
9127          * All errors are considered fatal in parsable mode.
9128          * NOMEM errors are always fatal, regardless of mode.
9129          * For other errors, we print diagnostics in human-readable
9130          * mode and processs what we can.
9131          */
9132         if (parsable || oferr == OFMT_ENOFIELDS) {
9133                 ofmt_close(ofmt);
9134                 die(buf);
9135         } else {
9136                 warn(buf);
9137         }
9138 }

9140 /*
9141  * Called from the walker dladm_walk_datalink_id() for each IB partition to
9142  * display IB partition specific information.
9143  */
9144 static dladm_status_t
9145 print_part(show_part_state_t *state, datalink_id_t linkid)
9146 {
9147         dladm_part_attr_t       attr;
9148         dladm_status_t          status;
9149         dladm_conf_t            conf;
9150         char                    part_over[MAXLINKNAMELEN];
9151         char                    part_name[MAXLINKNAMELEN];
9152         part_fields_buf_t       pbuf;
9153         boolean_t               force_in_conf = B_FALSE;

9155         /*
9156          * Get the information about the IB partition from the partition
9157          * datlink ID 'linkid'.
9158          */
9159         if ((status = dladm_part_info(handle, linkid, &attr, state->ps_flags))
9160             != DLADM_STATUS_OK)
9161                 return (status);

9163         /*
9164          * If an IB Phys link name was provided on the command line we have
9165          * the Phys link's datalink ID in the ps_over_id field of the state
9166          * structure. Proceed only if the IB partition represented by 'linkid'
9167          * was created over Phys link denoted by ps_over_id. The
9168          * 'dia_physlinkid' field of dladm_part_attr_t represents the IB Phys
9169          * link over which the partition was created.
9170          */
9171         if (state->ps_over_id != DATALINK_ALL_LINKID)
9172                 if (state->ps_over_id != attr.dia_physlinkid)
9173                         return (DLADM_STATUS_OK);

9175         /*
9176          * The linkid argument passed to this function is the datalink ID
9177          * of the IB Partition. Get the partitions name from this linkid.
9178          */
9179         if (dladm_datalink_id2info(handle, linkid, NULL, NULL,
9180             NULL, part_name, sizeof (part_name)) != DLADM_STATUS_OK)
```



```
9181                 return (DLADM_STATUS_BADARG);

9183         bzero(part_over, sizeof (part_over));

9185         /*
9186          * The 'dia_physlinkid' field contains the datalink ID of the IB Phys
9187          * link over which the partition was created. Use this linkid to get the
9188          * linkover field.
9189          */
9190         if (dladm_datalink_id2info(handle, attr.dia_physlinkid, NULL, NULL,
9191             NULL, part_over, sizeof (part_over)) != DLADM_STATUS_OK)
9192                 (void) sprintf(part_over, "?");
9193         state->ps_found = B_TRUE;

9195         /*
9196          * Read the FFORCE field from this datalink's persistent configuration
9197          * database line to determine if this datalink was created forcibly.
9198          * If this datalink is a temporary datalink, then it will not have an
9199          * entry in the persistent configuration, so check if force create flag
9200          * is set in the partition attributes.
9201          *
9202          * We need this two level check since persistent partitions brought up
9203          * by up-part during boot will have force create flag always set, since
9204          * we want up-part to always succeed even if the port is currently down
9205          * or P_Key is not yet available in the subnet.
9206          */
9207         if ((status = dladm_getsnap_conf(handle, linkid, &conf)) ==
9208             DLADM_STATUS_OK) {
9209                 (void) dladm_get_conf_field(handle, conf, FFORCE,
9210                     &force_in_conf, sizeof (boolean_t));
9211                 dladm_destroy_conf(handle, conf);
9212         } else if (status == DLADM_STATUS_NOTFOUND) {
9213                 /*
9214                  * for a temp link the force create flag will determine
9215                  * whether it was created with force flag.
9216                  */
9217                 force_in_conf = ((attr.dia_flags & DLADM_PART_FORCE_CREATE)
9218                     != 0);
9219         }

9221         (void) snprintf(pbuf.part_link, sizeof (pbuf.part_link),
9222             "%s", part_name);

9224         (void) snprintf(pbuf.part_over, sizeof (pbuf.part_over),
9225             "%s", part_over);

9227         (void) snprintf(pbuf.part_pkey, sizeof (pbuf.part_pkey),
9228             "%X", attr.dia_pkey);

9230         (void) get_linkstate(pbuf.part_link, B_TRUE, pbuf.part_state);

9232         (void) snprintf(pbuf.part_flags, sizeof (pbuf.part_flags),
9233             "%c----", force_in_conf ? 'f' : '-');

9235         ofmt_print(state->ps_ofmt, &pbuf);

9237         return (DLADM_STATUS_OK);
9238 }

9240 /* ARGSUSED */
9241 static int
9242 show_part(dladm_handle_t dh, datalink_id_t linkid, void *arg)
9243 {
9244         ((show_part_state_t *)arg)->ps_status = print_part(arg, linkid);
9245         return (DLADM_WALK_CONTINUE);
9246 }
```

```
9248 /*
9249  * Show the information about the IB partition objects.
9250  */
9251 static void
9252 do_show_part(int argc, char *argv[], const char *use)
9253 {
9254         int                     option;
9255         boolean_t               l_arg = B_FALSE;
9256         uint32_t                flags = DLADM_OPT_ACTIVE;
9257         datalink_id_t           linkid = DATALINK_ALL_LINKID;
9258         datalink_id_t           over_linkid = DATALINK_ALL_LINKID;
9259         char                    over_link[MAXLINKNAMELEN];
9260         show_part_state_t       state;
9261         dladm_status_t          status;
9262         boolean_t               o_arg = B_FALSE;
9263         char                    *fields_str = NULL;
9264         ofmt_handle_t           ofmt;
9265         ofmt_status_t           oferr;
9266         uint_t                  ofmtflags = 0;

9268         bzero(&state, sizeof (state));
9269         opterr = 0;
9270         while ((option = getopt_long(argc, argv, ":pPl:o:", show_part_lopts,
9271             NULL)) != -1) {
9272                 switch (option) {
9273                 case 'p':
9274                         state.ps_parsable = B_TRUE;
9275                         break;
9276                 case 'P':
9277                         flags = DLADM_OPT_PERSIST;
9278                         break;
9279                 case 'l':
9280                         /*
9281                          * The data link ID of the IB Phys link. When this
9282                          * argument is provided we list only the partition
9283                          * objects created over this IB Phys link.
9284                          */
9285                         if (strlcpy(over_link, optarg, MAXLINKNAMELEN) >=
9286                             MAXLINKNAMELEN)
9287                                 die("link name too long");

9289                         l_arg = B_TRUE;
9290                         break;
9291                 case 'o':
9292                         o_arg = B_TRUE;
9293                         fields_str = optarg;
9294                         break;
9295                 default:
9296                         die_opterr(optopt, option, use);
9297                 }
9298         }

9300         /*
9301          * Get the partition ID (optional last argument).
9302          */
9303         if (optind == (argc - 1)) {
9304                 status = dladm_name2info(handle, argv[optind], &linkid, NULL,
9305                     NULL, NULL);
9306                 if (status != DLADM_STATUS_OK) {
9307                         die_dlerr(status, "invalid partition link name '%s'",
9308                             argv[optind]);
9309                 }
9310                 (void) strlcpy(state.ps_part, argv[optind], MAXLINKNAMELEN);
9311         } else if (optind != argc) {
9312                 usage();
```



```
9313         }

9315         if (state.ps_parsable && !o_arg)
9316                 die("-p requires -o");

9318         /*
9319          * If an IB Phys link name was provided as an argument, then get its
9320          * datalink ID.
9321          */
9322         if (l_arg) {
9323                 status = dladm_name2info(handle, over_link, &over_linkid, NULL,
9324                     NULL, NULL);
9325                 if (status != DLADM_STATUS_OK) {
9326                         die_dlerr(status, "invalid link name '%s'", over_link);
9327                 }
9328         }

9330         state.ps_over_id = over_linkid; /* IB Phys link ID */
9331         state.ps_found = B_FALSE;
9332         state.ps_flags = flags;

9334         if (state.ps_parsable)
9335                 ofmtflags |= OFMT_PARSABLE;
9336         oferr = ofmt_open(fields_str, part_fields, ofmtflags, 0, &ofmt);
9337         dladm_ofmt_check(oferr, state.ps_parsable, ofmt);
9338         state.ps_ofmt = ofmt;

9340         /*
9341          * If a specific IB partition name was not provided as an argument,
9342          * walk all the datalinks and display the information for all
9343          * IB partitions. If IB Phys link was provided limit it to only
9344          * IB partitions created over that IB Phys link.
9345          */
9346         if (linkid == DATALINK_ALL_LINKID) {
9347                 (void) dladm_walk_datalink_id(show_part, handle, &state,
9348                     DATALINK_CLASS_PART, DATALINK_ANY_MEDIATYPE, flags);
9349         } else {
9350                 (void) show_part(handle, linkid, &state);
9351                 if (state.ps_status != DLADM_STATUS_OK) {
9352                         ofmt_close(ofmt);
9353                         die_dlerr(state.ps_status, "failed to show IB partition"
9354                             " '%s'", state.ps_part);
9355                 }
9356         }
9357         ofmt_close(ofmt);
9358 }


9361 /*
9362  * Called from the walker dladm_walk_datalink_id() for each IB Phys link to
9363  * display IB specific information for these Phys links.
9364  */
9365 static dladm_status_t
9366 print_ib(show_ib_state_t *state, datalink_id_t phys_linkid)
9367 {
9368         dladm_ib_attr_t         attr;
9369         dladm_status_t          status;
9370         char                    linkname[MAXLINKNAMELEN];
9371         char                    pkeystr[MAXPKEYLEN];
9372         int                     i;
9373         ib_fields_buf_t         ibuf;

9375         bzero(&attr, sizeof (attr));

9377         /*
9378          * Get the attributes of the IB Phys link from active/Persistent config
```

```
9379          * based on the flag passed.
9380          */
9381         if ((status = dladm_ib_info(handle, phys_linkid, &attr,
9382             state->is_flags)) != DLADM_STATUS_OK)
9383                 return (status);

9385         if ((state->is_link_id != DATALINK_ALL_LINKID) && (state->is_link_id
9386             != attr.dia_physlinkid)) {
9387                 dladm_free_ib_info(&attr);
9388                 return (DLADM_STATUS_OK);
9389         }

9391         /*
9392          * Get the data link name for the phys_linkid. If we are doing show-ib
9393          * for all IB Phys links, we have only the datalink IDs not the
9394          * datalink name.
9395          */
9396         if (dladm_datalink_id2info(handle, phys_linkid, NULL, NULL, NULL,
9397             linkname, MAXLINKNAMELEN) != DLADM_STATUS_OK)
9398                 return (status);

9400         (void) snprintf(ibuf.ib_link, sizeof (ibuf.ib_link),
9401             "%s", linkname);

9403         (void) snprintf(ibuf.ib_portnum, sizeof (ibuf.ib_portnum),
9404             "%d", attr.dia_portnum);

9406         (void) snprintf(ibuf.ib_hcaguid, sizeof (ibuf.ib_hcaguid),
9407             "%llX", attr.dia_hca_guid);

9409         (void) snprintf(ibuf.ib_portguid, sizeof (ibuf.ib_portguid),
9410             "%llX", attr.dia_port_guid);

9412         (void) get_linkstate(linkname, B_TRUE, ibuf.ib_state);

9414         /*
9415          * Create a comma separated list of pkeys from the pkey table returned
9416          * by the IP over IB driver instance.
9417          */
9418         bzero(ibuf.ib_pkeys, attr.dia_port_pkey_tbl_sz * sizeof (ib_pkey_t));
9419         for (i = 0; i < attr.dia_port_pkey_tbl_sz; i++) {
9420                 if (attr.dia_port_pkeys[i] != IB_PKEY_INVALID_FULL &&
9421                     attr.dia_port_pkeys[i] != IB_PKEY_INVALID_LIMITED) {
9422                         if (i == 0)
9423                                 (void) snprintf(pkeystr, MAXPKEYLEN, "%X",
9424                                     attr.dia_port_pkeys[i]);
9425                         else
9426                                 (void) snprintf(pkeystr, MAXPKEYLEN, ",%X",
9427                                     attr.dia_port_pkeys[i]);
9428                         (void) strlcat(ibuf.ib_pkeys, pkeystr, MAXPKEYSTRSZ);
9429                 }
9430         }

9432         dladm_free_ib_info(&attr);

9434         ofmt_print(state->is_ofmt, &ibuf);

9436         return (DLADM_STATUS_OK);
9437 }

9439 /* ARGSUSED */
9440 static int
9441 show_ib(dladm_handle_t dh, datalink_id_t linkid, void *arg)
9442 {
9443         ((show_ib_state_t *)arg)->is_status = print_ib(arg, linkid);
9444         return (DLADM_WALK_CONTINUE);
```



```
9445 }

9447 /*
9448  * Show the properties of one/all IB Phys links. This is different from
9449  * show-phys command since this will display IB specific information about the
9450  * Phys link like, HCA GUID, PORT GUID, PKEYS active for this port etc.
9451  */
9452 static void
9453 do_show_ib(int argc, char *argv[], const char *use)
9454 {
9455         int                     option;
9456         uint32_t                flags = DLADM_OPT_ACTIVE;
9457         datalink_id_t           linkid = DATALINK_ALL_LINKID;
9458         show_ib_state_t         state;
9459         dladm_status_t          status;
9460         boolean_t               o_arg = B_FALSE;
9461         char                    *fields_str = NULL;
9462         ofmt_handle_t           ofmt;
9463         ofmt_status_t           oferr;
9464         uint_t                  ofmtflags = 0;

9466         bzero(&state, sizeof (state));
9467         opterr = 0;
9468         while ((option = getopt_long(argc, argv, ":po:", show_lopts,
9469             NULL)) != -1) {
9470                 switch (option) {
9471                 case 'p':
9472                         state.is_parsable = B_TRUE;
9473                         break;
9474                 case 'o':
9475                         o_arg = B_TRUE;
9476                         fields_str = optarg;
9477                         break;
9478                 default:
9479                         die_opterr(optopt, option, use);
9480                 }
9481         }

9483         /* get IB Phys link ID (optional last argument) */
9484         if (optind == (argc - 1)) {
9485                 status = dladm_name2info(handle, argv[optind], &linkid, NULL,
9486                     NULL, NULL);
9487                 if (status != DLADM_STATUS_OK) {
9488                         die_dlerr(status, "invalid IB port name '%s'",
9489                             argv[optind]);
9490                 }
9491                 (void) strlcpy(state.is_link, argv[optind], MAXLINKNAMELEN);
9492         } else if (optind != argc) {
9493                 usage();
9494         }

9496         if (state.is_parsable && !o_arg)
9497                 die("-p requires -o");

9499         /*
9500          * linkid is the data link ID of the IB Phys link. By default it will
9501          * be DATALINK_ALL_LINKID.
9502          */
9503         state.is_link_id = linkid;
9504         state.is_flags = flags;

9506         if (state.is_parsable)
9507                 ofmtflags |= OFMT_PARSABLE;
9508         oferr = ofmt_open(fields_str, ib_fields, ofmtflags, 0, &ofmt);
9509         dladm_ofmt_check(oferr, state.is_parsable, ofmt);
9510         state.is_ofmt = ofmt;
```

```
9512         /*
9513          * If we are going to display the information for all IB Phys links
9514          * then we'll walk through all the datalinks for datalinks of Phys
9515          * class and media type IB.
9516          */
9517         if (linkid == DATALINK_ALL_LINKID) {
9518                 (void) dladm_walk_datalink_id(show_ib, handle, &state,
9519                     DATALINK_CLASS_PHYS, DL_IB, flags);
9520         } else {
9521                 /*
9522                  * We need to display the information only for the IB phys link
9523                  * linkid. Call show_ib for this link.
9524                  */
9525                 (void) show_ib(handle, linkid, &state);
9526                 if (state.is_status != DLADM_STATUS_OK) {
9527                         ofmt_close(ofmt);
9528                         die_dlerr(state.is_status, "failed to show IB Phys link"
9529                             " '%s'", state.is_link);
9530                 }
9531         }
9532         ofmt_close(ofmt);
9533 }

9535 /*
9536  * Create an IP over Infiniband partition object over an IB Phys link. The IB
9537  * Phys link is associated with an Infiniband HCA port. The IB partition object
9538  * is created over a port, pkey combination. This partition object represents
9539  * an instance of IP over IB interface.
9540  */
9541 /* ARGSUSED */
9542 static void
9543 do_create_part(int argc, char *argv[], const char *use)
9544 {
9545         int             status, option;
9546         int             flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
9547         char            *pname;
9548         char            *l_arg = NULL;
9549         char            *altroot = NULL;
9550         datalink_id_t   physlinkid = 0;
9551         datalink_id_t   partlinkid = 0;
9552         unsigned long   opt_pkey;
9553         ib_pkey_t       pkey = 0;
9554         char            *endp = NULL;
9555         char            propstr[DLADM_STRSIZE];
9556         dladm_arg_list_t        *proplist = NULL;

9558         propstr[0] = '\0';
9559         while ((option = getopt_long(argc, argv, ":tfl:P:R:p:",
9560             part_lopts, NULL)) != -1) {
9561                 switch (option) {
9562                 case 't':
9563                         /*
9564                          * Create a temporary IB partition object. This
9565                          * instance is not entered into the persistent database
9566                          * so it will not be recreated automatically on a
9567                          * reboot.
9568                          */
9569                         flags &= ~DLADM_OPT_PERSIST;
9570                         break;
9571                 case 'l':
9572                         /*
9573                          * The IB phys link over which the partition object will
9574                          * be created.
9575                          */
9576                         l_arg = optarg;
```



```
9577                         break;
9578                 case 'R':
9579                         altroot = optarg;
9580                         break;
9581                 case 'p':
9582                         (void) strlcat(propstr, optarg, DLADM_STRSIZE);
9583                         if (strlcat(propstr, ",", DLADM_STRSIZE) >=
9584                             DLADM_STRSIZE)
9585                                 die("property list too long '%s'", propstr);
9586                         break;
9587                 case 'P':
9588                         /*
9589                          * The P_Key for the port, pkey tuple of the partition
9590                          * object. This P_Key should exist in the IB subnet.
9591                          * The partition creation for a non-existent P_Key will
9592                          * fail unless the -f option is used.
9593                          *
9594                          * The P_Key is expected to be a hexadecimal number.
9595                          */
9596                         opt_pkey = strtoul(optarg, &endp, 16);
9597                         if (errno == ERANGE || opt_pkey > USHRT_MAX ||
9598                             *endp != '\0')
9599                                 die("Invalid pkey");

9601                         pkey = (ib_pkey_t)opt_pkey;
9602                         break;
9603                 case 'f':
9604                         flags |= DLADM_OPT_FORCE;
9605                         break;
9606                 default:
9607                         die_opterr(optopt, option, use);
9608                         break;
9609                 }
9610         }

9612         /* check required options */
9613         if (!l_arg)
9614                 usage();

9616         /* the partition name is a required operand */
9617         if (optind != (argc - 1))
9618                 usage();

9620         pname = argv[argc - 1];

9622         /*
9623          * Verify that the partition object's name is in the valid link name
9624          * format.
9625          */
9626         if (!dladm_valid_linkname(pname))
9627                 die("Invalid link name '%s'", pname);

9629         /* pkey is a mandatory argument */
9630         if (pkey == 0)
9631                 usage();

9633         if (altroot != NULL)
9634                 altroot_cmd(altroot, argc, argv);

9636         /*
9637          * Get the data link id of the IB Phys link over which we will be
9638          * creating partition object.
9639          */
9640         if (dladm_name2info(handle, l_arg,
9641             &physlinkid, NULL, NULL, NULL) != DLADM_STATUS_OK)
9642                 die("invalid link name '%s'", l_arg);
```

```
9644         /*
9645          * parse the property list provided with -p option.
9646          */
9647         if (dladm_parse_link_props(propstr, &proplist, B_FALSE)
9648             != DLADM_STATUS_OK)
9649                 die("invalid IB partition property");
9650
9651         /*
9652          * Call the library routine to create the partition object.
9653          */
9654         status = dladm_part_create(handle, physlinkid, pkey, flags, pname,
9655             &partlinkid, proplist);
9656         if (status != DLADM_STATUS_OK)
9657                 die_dlerr(status,
9658                     "partition %x creation over %s failed", pkey, l_arg);
9659 }
9660
9661 /*
9662  * Delete an IP over Infiniband partition object. The partition object should
9663  * be unplumbed before attempting the delete.
9664  */
9665 static void
9666 do_delete_part(int argc, char *argv[], const char *use)
9667 {
9668         int option, flags = DLADM_OPT_ACTIVE | DLADM_OPT_PERSIST;
9669         int status;
9670         char *altroot = NULL;
9671         datalink_id_t   partid;
9672
9673         opterr = 0;
9674         while ((option = getopt_long(argc, argv, "R:t", part_lopts,
9675             NULL)) != -1) {
9676                 switch (option) {
9677                 case 't':
9678                         flags &= ~DLADM_OPT_PERSIST;
9679                         break;
9680                 case 'R':
9681                         altroot = optarg;
9682                         break;
9683                 default:
9684                         die_opterr(optopt, option, use);
9685                 }
9686         }
9687
9688         /* get partition name (required last argument) */
9689         if (optind != (argc - 1))
9690                 usage();
9691
9692         if (altroot != NULL)
9693                 altroot_cmd(altroot, argc, argv);
9694
9695         /*
9696          * Get the data link id of the partition object given the partition
9697          * name.
9698          */
9699         status = dladm_name2info(handle, argv[optind], &partid, NULL, NULL,
9700             NULL);
9701         if (status != DLADM_STATUS_OK)
9702                 die("invalid link name '%s'", argv[optind]);
9703
9704         /*
9705          * Call the library routine to delete the IB partition. This will
9706          * result in the IB partition object and all its resources getting
9707          * deleted.
9708          */
```



```
9709         status = dladm_part_delete(handle, partid, flags);
9710         if (status != DLADM_STATUS_OK)
9711                 die_dlerr(status, "%s: partition deletion failed",
9712                     argv[optind]);
9713 }
9714
9715 /*
9716  * Bring up all or one IB partition already present in the persistent database
9717  * but not active yet.
9718  *
9719  * This sub-command is used during the system boot up to bring up all IB
9720  * partitions present in the persistent database. This is similar to a
9721  * create partition except that, the partitions are always created even if the
9722  * HCA port is down or P_Key is not present in the IB subnet. This is similar
9723  * to using the 'force' option while creating the partition except that the 'f'
9724  * flag will be set in the flags field only if the create-part for this command
9725  * was called with '-f' option.
9726  */
9727 /* ARGSUSED */
9728 static void
9729 do_up_part(int argc, char *argv[], const char *use)
9730 {
9731         datalink_id_t   partid = DATALINK_ALL_LINKID;
9732         dladm_status_t status;
9733
9734         /*
9735          * If a partition name was passed as an argument, get its data link
9736          * id. By default we'll attempt to bring up all IB partition data
9737          * links.
9738          */
9739         if (argc == 2) {
9740                 status = dladm_name2info(handle, argv[argc - 1], &partid, NULL,
9741                     NULL, NULL);
9742                 if (status != DLADM_STATUS_OK)
9743                         return;
9744         } else if (argc > 2) {
9745                 usage();
9746         }
9747
9748         (void) dladm_part_up(handle, partid, 0);
9749 }
```

```
**********************************************************
  110751 Sun Feb  9 05:30:59 2014
new/usr/src/man/man1m/dladm.1m
4585 dladm(1m) needs a 'help' subcommand
3755 dladm show-aggr documentation
3374 usage of 'dladm' does not match to its man page
**********************************************************
   1 '\" te
   2 .\" Copyright (c) 2008, Sun Microsystems, Inc. All Rights Reserved
   3 .\" Sun Microsystems, Inc. gratefully acknowledges The Open Group for permission
   4 .\" The Institute of Electrical and Electronics Engineers and The Open Group, ha
   5 .\" are reprinted and reproduced in electronic form in the Sun OS Reference Manu
   6 .\" and Electronics Engineers, Inc and The Open Group. In the event of any discr
   7 .\"  This notice shall appear on any product containing this material.
   8 .\" The contents of this file are subject to the terms of the Common Development
   9 .\" See the License for the specific language governing permissions and limitati
  10 .\" fields enclosed by brackets "[]" replaced with your own identifying informat
  11 .TH DLADM 1M "Feb 10, 2014"
  11 .TH DLADM 1M "Sep 23, 2009"
  12 .SH NAME
  13 dladm \- administer data links
  14 .SH SYNOPSIS
  15 .LP
  16 .nf
  17 \fBdladm\fR
  18 .fi

  20 .LP
  21 .nf
  22 #endif /* ! codereview */
  23 \fBdladm show-link\fR [\fB-P\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-p\fR
  24 \fBdladm rename-link\fR [\fB-R\fR \fIroot-dir\fR] \fIlink\fR \fInew-link\fR
  25 .fi

  27 .LP
  28 .nf
  29 \fBdladm delete-phys\fR \fIphys-link\fR
  30 \fBdladm show-phys\fR [\fB-P\fR] [\fB-m\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,..
  31 .fi

  33 .LP
  34 .nf
  35 \fBdladm create-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR \fIpolicy
  36      [\fB-T\fR \fItime\fR] [\fB-u\fR \fIaddress\fR] \fB-l\fR \fIether-link1\fR [
  37 \fBdladm modify-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR \fIpolicy
  38      [\fB-T\fR \fItime\fR] [\fB-u\fR \fIaddress\fR] \fIaggr-link\fR
  39 \fBdladm delete-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIaggr-link\fR
  40 \fBdladm add-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-link
  41      \fIaggr-link\fR
  42 \fBdladm remove-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-l
  43      \fIaggr-link\fR
  44 \fBdladm show-aggr\fR [\fB-PLx\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-p\
  45      [\fIaggr-link\fR]
  46 .fi

  48 .LP
  49 .nf
  50 \fBdladm create-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\f
  51      [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-d
  52      [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR
  53 .fi

  55 .LP
  56 .nf
  57 \fBdladm modify-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\f
  58      [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-d
```



```
  59      \fIbridge-name\fR
  60 .fi

  62 .LP
  63 .nf
  64 \fBdladm delete-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fIbridge-name\fR
  65 .fi

  67 .LP
  68 .nf
  69 \fBdladm add-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\fR \
  70 .fi

  72 .LP
  73 .nf
  74 \fBdladm remove-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\f
  75 .fi

  77 .LP
  78 .nf
  79 \fBdladm show-bridge\fR [\fB-flt\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-
  80      [\fIbridge-name\fR]
  81 .fi

  83 .LP
  84 .nf
  85 \fBdladm create-vlan\fR [\fB-ft\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-
  86 \fBdladm delete-vlan\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIvlan-link\fR
  87 \fBdladm show-vlan\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIvla
  88 .fi

  90 .LP
  91 .nf
  92 \fBdladm scan-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIwifi-link\fR]
  93 \fBdladm connect-wifi\fR [\fB-e\fR \fIessid\fR] [\fB-i\fR \fIbssid\fR] [\fB-k\fR
  94      [\fB-s\fR none | wep | wpa ] [\fB-a\fR open | shared] [\fB-b\fR bss | ibss]
  95      [\fB-m\fR a | b | g] [\fB-T\fR \fItime\fR] [\fIwifi-link\fR]
  96 \fBdladm disconnect-wifi\fR [\fB-a\fR] [\fIwifi-link\fR]
  97 \fBdladm show-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIwifi-link\fR]
  98 .fi

 100 .LP
 101 .nf
 102 \fBdladm show-ether\fR [\fB-x\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIet
 103 .fi

 105 .LP
 106 .nf
 107 \fBdladm set-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-p\fR \fIprop\f
 108 \fBdladm reset-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-p\fR \fIpro
 109 \fBdladm show-linkprop\fR [\fB-P\fR] [[\fB-c\fR] \fB-o\fR \fIfield\fR[,...]] [\f
 110 .fi

 112 .LP
 113 .nf
 114 \fBdladm create-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-f\fR \fIfile
 115 \fBdladm delete-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIsecobj\fR[,...
 116 \fBdladm show-secobj\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIs
 117 .fi

 119 .LP
 120 .nf
 121 \fBdladm create-vnic\fR [\fB-t\fR] \fB-l\fR \fIlink\fR [\fB-R\fR \fIroot-dir\fR]
 122      {factory \fB-n\fR \fIslot-identifier\fR]} | {random [\fB-r\fR \fIprefix\fR]
 123      [\fB-v\fR \fIvlan-id\fR] [\fB-p\fR \fIprop\fR=\fIvalue\fR[,...]] \fIvnic-li
 124 \fBdladm delete-vnic\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIvnic-link\fR
```

```
\fBdladm show-vnic\fR [\fB-pP\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [\fB-o\fR
     [\fB-l\fR \fIlink\fR] [\fIvnic-link\fR]
.fi

.LP
.nf
\fBdladm create-etherstub\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIetherstub\f
\fBdladm delete-etherstub\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIetherstub\f
\fBdladm show-etherstub\fR [\fIetherstub\fR]
.fi

.LP
.nf
\fBdladm create-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-T\fR \fItype\f
     \fIiptun-link\fR
\fBdladm modify-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-s\fR \fItsrc\
\fBdladm delete-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIiptun-link\fR
\fBdladm show-iptun\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIip
.fi

.LP
.nf
\fBdladm show-usage\fR [\fB-a\fR] \fB-f\fR \fIfilename\fR [\fB-p\fR \fIplotfile\
     [\fB-e\fR \fItime\fR] [\fIlink\fR]
.fi

.LP
.nf
\fBdladm help\fR [\fIsubcommand\fR]
.fi

#endif /* ! codereview */
.SH DESCRIPTION
.sp
.LP
The \fBdladm\fR command is used to administer data-links. A data-link is
represented in the system as a \fBSTREAMS DLPI\fR (v2) interface which can be
plumbed under protocol stacks such as \fBTCP/IP\fR. Each data-link relies on
either a single network device or an aggregation of devices to send packets to
or receive packets from a network.
.sp
.LP
Each \fBdladm\fR subcommand operates on one of the following objects:
.sp
.ne 2
.na
\fB\fBlink\fR\fR
.ad
.sp .6
.RS 4n
A datalink, identified by a name. In general, the name can use any alphanumeric
characters (or the underscore, \fB_\fR), but must start with an alphabetic
character and end with a number. A datalink name can be at most 31 characters,
and the ending number must be between 0 and 4294967294 (inclusive). The ending
number must not begin with a zero. Datalink names between 3 and 8 characters
are recommended.
.sp
Some subcommands operate only on certain types or classes of datalinks. For
those cases, the following object names are used:
.sp
.ne 2
.na
\fB\fBphys-link\fR\fR
.ad
.sp .6
.RS 4n
```



```
A physical datalink.
.RE

.sp
.ne 2
.na
\fB\fBvlan-link\fR\fR
.ad
.sp .6
.RS 4n
A VLAN datalink.
.RE

.sp
.ne 2
.na
\fB\fBaggr-link\fR\fR
.ad
.sp .6
.RS 4n
An aggregation datalink (or a key; see NOTES).
.RE

.sp
.ne 2
.na
\fB\fBether-link\fR\fR
.ad
.sp .6
.RS 4n
A physical Ethernet datalink.
.RE

.sp
.ne 2
.na
\fB\fBwifi-link\fR\fR
.ad
.sp .6
.RS 4n
A WiFi datalink.
.RE

.sp
.ne 2
.na
\fB\fBvnic-link\fR\fR
.ad
.sp .6
.RS 4n
A virtual network interface created on a link or an \fBetherstub\fR. It is a
pseudo device that can be treated as if it were an network interface card on a
machine.
.RE

.sp
.ne 2
.na
\fB\fBiptun-link\fR\fR
.ad
.sp .6
.RS 4n
An IP tunnel link.
.RE

.RE
```

```
.sp
.ne 2
.na
\fB\fBdev\fR\fR
.ad
.sp .6
.RS 4n
A network device, identified by concatenation of a driver name and an instance
number.
.RE

.sp
.ne 2
.na
\fB\fBetherstub\fR\fR
.ad
.sp .6
.RS 4n
An Ethernet stub can be used instead of a physical NIC to create VNICs. VNICs
created on an \fBetherstub\fR will appear to be connected through a virtual
switch, allowing complete virtual networks to be built without physical
hardware.
.RE

.sp
.ne 2
.na
\fB\fBbridge\fR\fR
.ad
.sp .6
.RS 4n
A bridge instance, identified by an administratively-chosen name. The name may
use any alphanumeric characters or the underscore, \fB_\fR, but must start and
end with an alphabetic character. A bridge name can be at most 31 characters.
The name \fBdefault\fR is reserved, as are all names starting with \fBSUNW\fR.
.sp
Note that appending a zero (\fB0\fR) to a bridge name produces a valid link
name, used for observability.
.RE

.sp
.ne 2
.na
\fB\fBsecobj\fR\fR
.ad
.sp .6
.RS 4n
A secure object, identified by an administratively-chosen name. The name can
use any alphanumeric characters, as well as underscore (\fB_\fR), period
(\fB\&.\fR), and hyphen (\fB-\fR). A secure object name can be at most 32
characters.
.RE

.SS "Options"
.sp
.LP
Each \fBdladm\fR subcommand has its own set of options. However, many of the
subcommands have the following as a common option:
.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
```



```
Specifies an alternate root directory where the operation-such as creation,
deletion, or renaming-should apply.
.RE

.SS "SUBCOMMANDS"
.sp
.LP
The following subcommands are supported:
.sp
.ne 2
.na
\fB\fBdladm show-link\fR [\fB-P\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
[[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]][\fIlink\fR]\fR
.ad
.sp .6
.RS 4n
Show link configuration information (the default) or statistics, either for all
datalinks or for the specified link \fIlink\fR. By default, the system is
configured with one datalink for each known network device.
.sp
.ne 2
.na
\fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
.ad
.sp .6
.RS 4n
A case-insensitive, comma-separated list of output fields to display. When not
modified by the \fB-s\fR option (described below), the field name must be one
of the fields listed below, or the special value \fBall\fR to display all
fields. By default (without \fB-o\fR), \fBshow-link\fR displays all fields.
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the datalink.
.RE

.sp
.ne 2
.na
\fB\fBCLASS\fR\fR
.ad
.sp .6
.RS 4n
The class of the datalink. \fBdladm\fR distinguishes between the following
classes:
.sp
.ne 2
.na
\fB\fBphys\fR\fR
.ad
.sp .6
.RS 4n
A physical datalink. The \fBshow-phys\fR subcommand displays more detail for
this class of datalink.
.RE

.sp
.ne 2
.na
\fB\fBaggr\fR\fR
.ad
.sp .6
```

```
.RS 4n
An IEEE 802.3ad link aggregation. The \fBshow-aggr\fR subcommand displays more
detail for this class of datalink.
.RE

.sp
.ne 2
.na
\fB\fBvlan\fR\fR
.ad
.sp .6
.RS 4n
A VLAN datalink. The \fBshow-vlan\fR subcommand displays more detail for this
class of datalink.
.RE

.sp
.ne 2
.na
\fB\fBvnic\fR\fR
.ad
.sp .6
.RS 4n
A virtual network interface. The \fBshow-vnic\fR subcommand displays more
detail for this class of datalink.
.RE

.RE

.sp
.ne 2
.na
\fB\fBMTU\fR\fR
.ad
.sp .6
.RS 4n
The maximum transmission unit size for the datalink being displayed.
.RE

.sp
.ne 2
.na
\fB\fBSTATE\fR\fR
.ad
.sp .6
.RS 4n
The link state of the datalink. The state can be \fBup\fR, \fBdown\fR, or
\fBunknown\fR.
.RE

.sp
.ne 2
.na
\fB\fBBRIDGE\fR\fR
.ad
.sp .6
.RS 4n
The name of the bridge to which this link is assigned, if any.
.RE

.sp
.ne 2
.na
\fB\fBOVER\fR\fR
.ad
.sp .6
```



```
.RS 4n
The physical datalink(s) over which the datalink is operating. This applies to
\fBaggr\fR, \fBbridge\fR, and \fBvlan\fR classes of datalinks. A VLAN is
created over a single physical datalink, a bridge has multiple attached links,
and an aggregation is comprised of one or more physical datalinks.
.RE

When the \fB-o\fR option is used in conjunction with the \fB-s\fR option, used
to display link statistics, the field name must be one of the fields listed
below, or the special value \fBall\fR to display all fields
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the datalink.
.RE

.sp
.ne 2
.na
\fB\fBIPACKETS\fR\fR
.ad
.sp .6
.RS 4n
Number of packets received on this link.
.RE

.sp
.ne 2
.na
\fB\fBRBYTES\fR\fR
.ad
.sp .6
.RS 4n
Number of bytes received on this link.
.RE

.sp
.ne 2
.na
\fB\fBIERRORS\fR\fR
.ad
.sp .6
.RS 4n
Number of input errors.
.RE

.sp
.ne 2
.na
\fB\fBOPACKETS\fR\fR
.ad
.sp .6
.RS 4n
Number of packets sent on this link.
.RE

.sp
.ne 2
.na
\fB\fBOBYTES\fR\fR
.ad
.sp .6
```

```
.RS 4n
Number of bytes sent on this link.
Number of bytes received on this link.
.RE

.sp
.ne 2
.na
\fB\fBOERRORS\fR\fR
.ad
.sp .6
.RS 4n
Number of output errors.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-p\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Display using a stable machine-parseable format. The \fB-o\fR option is
required with \fB-p\fR. See "Parseable Output Format", below.
.RE

.sp
.ne 2
.na
\fB\fB-P\fR, \fB--persistent\fR\fR
.ad
.sp .6
.RS 4n
Display the persistent link configuration.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR, \fB--statistics\fR\fR
.ad
.sp .6
.RS 4n
Display link statistics.
.RE

.sp
.ne 2
.na
\fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
.ad
.sp .6
.RS 4n
Used with the \fB-s\fR option to specify an interval, in seconds, at which
statistics should be displayed. If this option is not specified, statistics
will be displayed only once.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm rename-link\fR [\fB-R\fR \fIroot-dir\fR] \fIlink\fR
```



```
\fInew-link\fR\fR
.ad
.sp .6
.RS 4n
Rename \fIlink\fR to \fInew-link\fR. This is used to give a link a meaningful
name, or to associate existing link configuration such as link properties of a
removed device with a new device. See the \fBEXAMPLES\fR section for specific
examples of how this subcommand is used.
.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm delete-phys\fR \fIphys-link\fR\fR
.ad
.sp .6
.RS 4n
This command is used to delete the persistent configuration of a link
associated with physical hardware which has been removed from the system. See
the \fBEXAMPLES\fR section.
.RE

.sp
.ne 2
.na
\fB\fBdladm show-phys\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
[\fB-H\fR] [\fIphys-link\fR]\fR
.ad
.sp .6
.RS 4n
Show the physical device and attributes of all physical links, or of the named
physical link. Without \fB-P\fR, only physical links that are available on the
running system are displayed.
.sp
.ne 2
.na
\fB\fB-H\fR\fR
.ad
.sp .6
.RS 4n
Show hardware resource usage, as returned by the NIC driver. Output from
\fB-H\fR displays the following elements:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
A physical device corresponding to a NIC driver.
.RE

.sp
.ne 2
.na
\fB\fBGROUP\fR\fR
```

```
652 .ad
653 .sp .6
654 .RS 4n
655 A collection of rings.
656 .RE

658 .sp
659 .ne 2
660 .na
661 \fB\fBGROUPTYPE\fR\fR
662 .ad
663 .sp .6
664 .RS 4n
665 RX or TX. All rings in a group are of the same group type.
666 .RE

668 .sp
669 .ne 2
670 .na
671 \fB\fBRINGS\fR\fR
672 .ad
673 .sp .6
674 .RS 4n
675 A hardware resource used by a data link, subject to assignment by a driver to
676 different groups.
677 .RE

679 .sp
680 .ne 2
681 .na
682 \fB\fBCLIENTS\fR\fR
683 .ad
684 .sp .6
685 .RS 4n
686 MAC clients that are using the rings within a group.
687 .RE

689 .RE

691 .sp
692 .ne 2
693 .na
694 \fB\fB-o\fR \fIfield\fR, \fB--output\fR=\fIfield\fR\fR
695 .ad
696 .sp .6
697 .RS 4n
698 A case-insensitive, comma-separated list of output fields to display. The field
699 name must be one of the fields listed below, or the special value \fBall\fR, to
700 display all fields. For each link, the following fields can be displayed:
701 .sp
702 .ne 2
703 .na
704 \fB\fBLINK\fR\fR
705 .ad
706 .sp .6
707 .RS 4n
708 The name of the datalink.
709 .RE

711 .sp
712 .ne 2
713 .na
714 \fB\fBMEDIA\fR\fR
715 .ad
716 .sp .6
717 .RS 4n
```



```
718 The media type provided by the physical datalink.
719 .RE

721 .sp
722 .ne 2
723 .na
724 \fB\fBSTATE\fR\fR
725 .ad
726 .sp .6
727 .RS 4n
728 The state of the link. This can be \fBup\fR, \fBdown\fR, or \fBunknown\fR.
729 .RE

731 .sp
732 .ne 2
733 .na
734 \fB\fBSPEED\fR\fR
735 .ad
736 .sp .6
737 .RS 4n
738 The current speed of the link, in megabits per second.
739 .RE

741 .sp
742 .ne 2
743 .na
744 \fB\fBDUPLEX\fR\fR
745 .ad
746 .sp .6
747 .RS 4n
748 For Ethernet links, the full/half duplex status of the link is displayed if the
749 link state is \fBup\fR. The duplex is displayed as \fBunknown\fR in all other
750 cases.
751 .RE

753 .sp
754 .ne 2
755 .na
756 \fB\fBDEVICE\fR\fR
757 .ad
758 .sp .6
759 .RS 4n
760 The name of the physical device under this link.
761 .RE

763 .RE

765 .sp
766 .ne 2
767 .na
768 \fB\fB-p\fR, \fB--parseable\fR\fR
769 .ad
770 .sp .6
771 .RS 4n
772 Display using a stable machine-parseable format. The \fB-o\fR option is
773 required with \fB-p\fR. See "Parseable Output Format", below.
774 .RE

776 .sp
777 .ne 2
778 .na
779 \fB\fB-P\fR, \fB--persistent\fR\fR
780 .ad
781 .sp .6
782 .RS 4n
783 This option displays persistent configuration for all links, including those
```

that have been removed from the system. The output provides a \fBFLAGS\fR column in which the \fBr\fR flag indicates that the physical device associated with a physical link has been removed. For such links, \fBdelete-phys\fR can be used to purge the link’s configuration from the system.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm create-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR \fIpolicy\fR] [\fB-L\fR \fImode\fR] [\fB-T\fR \fItime\fR] [\fB-u\fR \fIaddress\fR] \fB-l\fR \fIether-link1\fR [\fB-l\fR \fIether-link2\fR...] \fIaggr-link\fR\fR
.ad
.sp .6
.RS 4n
Combine a set of links into a single IEEE 802.3ad link aggregation named \fIaggr-link\fR. The use of an integer \fIkey\fR to generate a link name for the aggregation is also supported for backward compatibility. Many of the \fB*\fR\fB-aggr\fR subcommands below also support the use of a \fIkey\fR to refer to a given aggregation, but use of the aggregation link name is preferred. See the \fBNOTES\fR section for more information on keys.
.sp
\fBdladm\fR supports a number of port selection policies for an aggregation of ports. (See the description of the \fB-P\fR option, below.) If you do not specify a policy, \fBcreate-aggr\fR uses the default, the L4 policy, described under the \fB-P\fR option.
.sp
.ne 2
.na
\fB\fB-l\fR \fIether-link\fR, \fB--link\fR=\fIether-link\fR\fR
.ad
.sp .6
.RS 4n
Each Ethernet link (or port) in the aggregation is specified using an \fB-l\fR option followed by the name of the link to be included in the aggregation. Multiple links are included in the aggregation by specifying multiple \fB-l\fR options. For backward compatibility with previous versions of Solaris, the \fBdladm\fR command also supports the using the \fB-d\fR option (or \fB--dev\fR) with a device name to specify links by their underlying device name. The other \fB*\fR\fB-aggr\fR subcommands that take \fB-l\fRoptions also accept \fB-d\fR.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the aggregation is temporary. Temporary aggregations last until the next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-P\fR \fIpolicy\fR, \fB--policy\fR=\fIpolicy\fR\fR
.ad
.br
.na
\fB\fR
.ad
.sp .6
.RS 4n
Specifies the port selection policy to use for load spreading of outbound traffic. The policy specifies which \fIdev\fR object is used to send packets. A policy is a list of one or more layers specifiers separated by commas. A layer specifier is one of the following:
.sp
.ne 2
.na
\fB\fBL2\fR\fR
.ad
.sp .6
.RS 4n
Select outbound device according to source and destination \fBMAC\fR addresses of the packet.
.RE

.sp
.ne 2
.na
\fB\fBL3\fR\fR
.ad
.sp .6
.RS 4n
Select outbound device according to source and destination \fBIP\fR addresses of the packet.
.RE

.sp
.ne 2
.na
\fB\fBL4\fR\fR
.ad
.sp .6
.RS 4n
Select outbound device according to the upper layer protocol information contained in the packet. For \fBTCP\fR and \fBUDP\fR, this includes source and destination ports. For IPsec, this includes the \fBSPI\fR (Security Parameters Index).
.RE

For example, to use upper layer protocol information, the following policy can be used:
.sp
.in +2
.nf
-P L4
.fi
.in -2
.sp

Note that policy L4 is the default.
.sp
To use the source and destination \fBMAC\fR addresses as well as the source and destination \fBIP\fR addresses, the following policy can be used:
.sp

```
.in +2
.nf
-P L2,L3
.fi
.in -2
.sp

.RE

.sp
.ne 2
.na
\fB\fB-L\fR \fImode\fR, \fB--lacp-mode\fR=\fImode\fR\fR
.ad
.sp .6
.RS 4n
Specifies whether \fBLACP\fR should be used and, if used, the mode in which it
should operate. Supported values are \fBoff\fR, \fBactive\fR or \fBpassive\fR.
.RE

.sp
.ne 2
.na
\fB\fB-T\fR \fItime\fR, \fB--lacp-timer\fR=\fItime\fR\fR
.ad
.br
.na
\fB\fR
.ad
.sp .6
.RS 4n
Specifies the \fBLACP\fR timer value. The supported values are \fBshort\fR or
\fBlong\fRjjj.
.RE

.sp
.ne 2
.na
\fB\fB-u\fR \fIaddress\fR, \fB--unicast\fR=\fIaddress\fR\fR
.ad
.sp .6
.RS 4n
Specifies a fixed unicast hardware address to be used for the aggregation. If
this option is not specified, then an address is automatically chosen from the
set of addresses of the component devices.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm modify-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR
\fIpolicy\fR] [\fB-L\fR \fImode\fR] [\fB-T\fR \fItime\fR] [\fB-u\fR
\fIaddress\fR] \fIaggr-link\fR\fR
.ad
.sp .6
.RS 4n
Modify the parameters of the specified aggregation.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
```



```
Specifies that the modification is temporary. Temporary aggregations last until
the next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-P\fR \fIpolicy\fR, \fB--policy\fR=\fIpolicy\fR\fR
.ad
.sp .6
.RS 4n
Specifies the port selection policy to use for load spreading of outbound
traffic. See \fBdladm create-aggr\fR for a description of valid policy values.
.RE

.sp
.ne 2
.na
\fB\fB-L\fR \fImode\fR, \fB--lacp-mode\fR=\fImode\fR\fR
.ad
.sp .6
.RS 4n
Specifies whether \fBLACP\fR should be used and, if used, the mode in which it
should operate. Supported values are \fBoff\fR, \fBactive\fR, or \fBpassive\fR.
.RE

.sp
.ne 2
.na
\fB\fB-T\fR \fItime\fR, \fB--lacp-timer\fR=\fItime\fR\fR
.ad
.br
.na
\fB\fR
.ad
.sp .6
.RS 4n
Specifies the \fBLACP\fR timer value. The supported values are \fBshort\fR or
\fBlong\fR.
.RE

.sp
.ne 2
.na
\fB\fB-u\fR \fIaddress\fR, \fB--unicast\fR=\fIaddress\fR\fR
.ad
.sp .6
.RS 4n
Specifies a fixed unicast hardware address to be used for the aggregation. If
this option is not specified, then an address is automatically chosen from the
set of addresses of the component devices.
.RE

.RE

.sp
```

```
.ne 2
.na
\fB\fBdladm delete-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIaggr-link\fR\fR
.ad
.sp .6
.RS 4n
Deletes the specified aggregation.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the deletion is temporary. Temporary deletions last until the
next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm add-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
\fIether-link1\fR [\fB--link\fR=\fIether-link2\fR...] \fIaggr-link\fR\fR
.ad
.sp .6
.RS 4n
Adds links to the specified aggregation.
.sp
.ne 2
.na
\fB\fB-l\fR \fIether-link\fR, \fB--link\fR=\fIether-link\fR\fR
.ad
.sp .6
.RS 4n
Specifies an Ethernet link to add to the aggregation. Multiple links can be
added by supplying multiple \fB-l\fR options.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the additions are temporary. Temporary additions last until the
next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
```



```
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm remove-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
\fIether-link1\fR [\fB--l\fR=\fIether-link2\fR...] \fIaggr-link\fR\fR
.ad
.sp .6
.RS 4n
Removes links from the specified aggregation.
.sp
.ne 2
.na
\fB\fB-l\fR \fIether-link\fR, \fB--link\fR=\fIether-link\fR\fR
.ad
.sp .6
.RS 4n
Specifies an Ethernet link to remove from the aggregation. Multiple links can
be added by supplying multiple \fB-l\fR options.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the removals are temporary. Temporary removal last until the
next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm show-aggr\fR [\fB-PLx\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
[[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIaggr-link\fR]\fR
.ad
.sp .6
.RS 4n
Show aggregation configuration (the default), \fBLACP\fR information, or
statistics, either for all aggregations or for the specified aggregation.
.sp
By default (with no options), the following fields can be displayed:
.sp
.ne 2
.na
```

```
1180 \fB\fBLINK\fR\fR
1181 .ad
1182 .sp .6
1183 .RS 4n
1184 The name of the aggregation link.
1185 .RE

1187 .sp
1188 .ne 2
1189 .na
1190 \fB\fBPOLICY\fR\fR
1191 .ad
1192 .sp .6
1193 .RS 4n
1194 The LACP policy of the aggregation. See the \fBcreate-aggr\fR \fB-P\fR option
1195 for a description of the possible values.
1196 .RE

1198 .sp
1199 .ne 2
1200 .na
1201 \fB\fBADDRPOLICY\fR\fR
1202 .ad
1203 .sp .6
1204 .RS 4n
1205 Either \fBauto\fR, if the aggregation is configured to automatically configure
1206 its unicast MAC address (the default if the \fB-u\fR option was not used to
1207 create or modify the aggregation), or \fBfixed\fR, if \fB-u\fR was used to set
1208 a fixed MAC address.
1209 .RE

1211 .sp
1212 .ne 2
1213 .na
1214 \fB\fBLACPACTIVITY\fR\fR
1215 .ad
1216 .sp .6
1217 .RS 4n
1218 The LACP mode of the aggregation. Possible values are \fBoff\fR, \fBactive\fR,
1219 or \fBpassive\fR, as set by the \fB-l\fR option to \fBcreate-aggr\fR or
1220 \fBmodify-aggr\fR.
1221 .RE

1223 .sp
1224 .ne 2
1225 .na
1226 \fB\fBLACPTIMER\fR\fR
1227 .ad
1228 .sp .6
1229 .RS 4n
1230 The LACP timer value of the aggregation as set by the \fB-T\fR option of
1231 \fBcreate-aggr\fR or \fBmodify-aggr\fR.
1232 .RE

1234 .sp
1235 .ne 2
1236 .na
1237 \fB\fBFLAGS\fR\fR
1238 .ad
1239 .sp .6
1240 .RS 4n
1241 A set of state flags associated with the aggregation. The only possible flag is
1242 \fBf\fR, which is displayed if the administrator forced the creation the
1243 aggregation using the \fB-f\fR option to \fBcreate-aggr\fR. Other flags might
1244 be defined in the future.
1245 .RE
```



```
1247 The \fBshow-aggr\fR command accepts the following options:
1248 .sp
1249 .ne 2
1250 .na
1251 \fB\fB-L\fR, \fB--lacp\fR\fR
1252 .ad
1253 .sp .6
1254 .RS 4n
1255 Displays detailed \fBLACP\fR information for the aggregation link and each
1256 underlying port. Most of the state information displayed by this option is
1257 defined by IEEE 802.3. With this option, the following fields can be displayed:
1258 .sp
1259 .ne 2
1260 .na
1261 \fB\fBLINK\fR\fR
1262 .ad
1263 .sp .6
1264 .RS 4n
1265 The name of the aggregation link.
1266 .RE

1268 .sp
1269 .ne 2
1270 .na
1271 \fB\fBPORT\fR\fR
1272 .ad
1273 .sp .6
1274 .RS 4n
1275 The name of one of the underlying aggregation ports.
1276 .RE

1278 .sp
1279 .ne 2
1280 .na
1281 \fB\fBAGGREGATABLE\fR\fR
1282 .ad
1283 .sp .6
1284 .RS 4n
1285 Whether the port can be added to the aggregation.
1286 .RE

1288 .sp
1289 .ne 2
1290 .na
1291 \fB\fBSYNC\fR\fR
1292 .ad
1293 .sp .6
1294 .RS 4n
1295 If \fByes\fR, the system considers the port to be synchronized and part of the
1296 aggregation.
1297 .RE

1299 .sp
1300 .ne 2
1301 .na
1302 \fB\fBCOLL\fR\fR
1303 .ad
1304 .sp .6
1305 .RS 4n
1306 If \fByes\fR, collection of incoming frames is enabled on the associated port.
1307 .RE

1309 .sp
1310 .ne 2
1311 .na
```

```
\fB\fBDIST\fR\fR
.ad
.sp .6
.RS 4n
If \fByes\fR, distribution of outgoing frames is enabled on the associated
port.
.RE

.sp
.ne 2
.na
\fB\fBDEFAULTED\fR\fR
.ad
.sp .6
.RS 4n
If \fByes\fR, the port is using defaulted partner information (that is, has not
received LACP data from the LACP partner).
.RE

.sp
.ne 2
.na
\fB\fBEXPIRED\fR\fR
.ad
.sp .6
.RS 4n
If \fByes\fR, the receive state of the port is in the \fBEXPIRED\fR state.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-x\fR, \fB--extended\fR\fR
.ad
.sp .6
.RS 4n
Display additional aggregation information including detailed information on
each underlying port. With \fB-x\fR, the following fields can be displayed:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the aggregation link.
.RE

.sp
.ne 2
.na
\fB\fBPORT\fR\fR
.ad
.sp .6
.RS 4n
The name of one of the underlying aggregation ports.
.RE

.sp
.ne 2
.na
\fB\fBSPEED\fR\fR
.ad
.sp .6
```



```
.RS 4n
The speed of the link or port in megabits per second.
.RE

.sp
.ne 2
.na
\fB\fBDUPLEX\fR\fR
.ad
.sp .6
.RS 4n
The full/half duplex status of the link or port is displayed if the link state
is \fBup\fR. The duplex status is displayed as \fBunknown\fR in all other
cases.
.RE

.sp
.ne 2
.na
\fB\fBSTATE\fR\fR
.ad
.sp .6
.RS 4n
The link state. This can be \fBup\fR, \fBdown\fR, or \fBunknown\fR.
.RE

.sp
.ne 2
.na
\fB\fBADDRESS\fR\fR
.ad
.sp .6
.RS 4n
The MAC address of the link or port.
.RE

.sp
.ne 2
.na
\fB\fBPORTSTATE\fR\fR
.ad
.sp .6
.RS 4n
This indicates whether the individual aggregation port is in the \fBstandby\fR
or \fBattached\fR state.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
.ad
.sp .6
.RS 4n
A case-insensitive, comma-separated list of output fields to display. The field
name must be one of the fields listed above, or the special value \fBall\fR, to
display all fields. The fields applicable to the \fB-o\fR option are limited to
those listed under each output mode. For example, if using \fB-L\fR, only the
fields listed under \fB-L\fR, above, can be used with \fB-o\fR.
.RE

.sp
.ne 2
.na
```

\fB\fB-p\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Display using a stable machine-parseable format. The \fB-o\fR option is required with \fB-p\fR. See "Parseable Output Format", below.
.RE

.sp
.ne 2
.na
\fB\fB-P\fR, \fB--persistent\fR\fR
.ad
.sp .6
.RS 4n
Display the persistent aggregation configuration rather than the state of the running system.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR, \fB--statistics\fR\fR
.ad
.sp .6
.RS 4n
Displays aggregation statistics.
.RE

.sp
.ne 2
.na
\fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
.ad
.sp .6
.RS 4n
Used with the \fB-s\fR option to specify an interval, in seconds, at which statistics should be displayed. If this option is not specified, statistics will be displayed only once.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm create-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-p\fR \fIpriority\fR] [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-delay\fR] [\fB-f\fR \fIforce-protocol\fR] [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR\fR
.ad
.sp .6
.RS 4n
Create an 802.1D bridge instance and optionally assign one or more network links to the new bridge. By default, no bridge instances are present on the system.
.sp
In order to bridge between links, you must create at least one bridge instance. Each bridge instance is separate, and there is no forwarding connection between bridges.
.sp
.ne 2
.na
\fB\fB-P\fR \fIprotect\fR, \fB--protect\fR=\fIprotect\fR\fR
.ad
.sp .6

.RS 4n
Specifies a protection method. The defined protection methods are \fBstp\fR for the Spanning Tree Protocol and trill for \fBTRILL\fR, which is used on RBridges. The default value is \fBstp\fR.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIpriority\fR, \fB--priority\fR=\fIpriority\fR\fR
.ad
.sp .6
.RS 4n
Specifies the Bridge Priority. This sets the IEEE STP priority value for determining the root bridge node in the network. The default value is \fB32768\fR. Valid values are \fB0\fR (highest priority) to \fB61440\fR (lowest priority), in increments of 4096.
.sp
If a value not evenly divisible by 4096 is used, the system silently rounds downward to the next lower value that is divisible by 4096.
.RE

.sp
.ne 2
.na
\fB\fB-m\fR \fImax-age\fR, \fB--max-age\fR=\fImax-age\fR\fR
.ad
.sp .6
.RS 4n
Specifies the maximum age for configuration information in seconds. This sets the STP Bridge Max Age parameter. This value is used for all nodes in the network if this node is the root bridge. Bridge link information older than this time is discarded. It defaults to 20 seconds. Valid values are from 6 to 40 seconds. See the \fB-d\fR \fIforward-delay\fR parameter for additional constraints.
.RE

.sp
.ne 2
.na
\fB\fB-h\fR \fIhello-time\fR, \fB--hello-time\fR=\fIhello-time\fR\fR
.ad
.sp .6
.RS 4n
Specifies the STP Bridge Hello Time parameter. When this node is the root node, it sends Configuration BPDUs at this interval throughout the network. The default value is 2 seconds. Valid values are from 1 to 10 seconds. See the \fB-d\fR \fIforward-delay\fR parameter for additional constraints.
.RE

.sp
.ne 2
.na
\fB\fB-d\fR \fIforward-delay\fR, \fB--forward-delay\fR=\fIforward-delay\fR\fR
.ad
.sp .6

.RS 4n
Specifies the STP Bridge Forward Delay parameter. When this node is the root node, then all bridges in the network use this timer to sequence the link states when a port is enabled. The default value is 15 seconds. Valid values are from 4 to 30 seconds.
.sp
Bridges must obey the following two constraints:
.sp
.in +2
.nf
2 * (\fIforward-delay\fR - 1.0) >= \fImax-age\fR

\fImax-age\fR >= 2 * (\fIhello-time\fR + 1.0)
.fi
.in -2
.sp

Any parameter setting that would violate those constraints is treated as an error and causes the command to fail with a diagnostic message. The message provides valid alternatives to the supplied values.
.RE

.sp
.ne 2
.na
\fB\fB-f\fR \fIforce-protocol\fR,
\fB--force-protocol\fR=\fIforce-protocol\fR\fR
.ad
.sp .6
.RS 4n
Specifies the MSTP forced maximum supported protocol. The default value is 3. Valid values are non-negative integers. The current implementation does not support RSTP or MSTP, so this currently has no effect. However, to prevent MSTP from being used in the future, the parameter may be set to \fB0\fR for STP only or \fB2\fR for STP and RSTP.
.RE

.sp
.ne 2
.na
\fB\fB-l\fR \fIlink\fR, \fB--link\fR=\fIlink\fR\fR
.ad
.sp .6
.RS 4n
Specifies one or more links to add to the newly-created bridge. This is similar to creating the bridge and then adding one or more links, as with the \fBadd-bridge\fR subcommand. However, if any of the links cannot be added, the entire command fails, and the new bridge itself is not created. To add multiple links on the same command line, repeat this option for each link. You are permitted to create bridges without links. For more information about link assignments, see the \fBadd-bridge\fR subcommand.
.RE

Bridge creation and link assignment require the \fBPRIV_SYS_DL_CONFIG\fR privilege. Bridge creation might fail if the optional bridging feature is not installed on the system.
.RE

.sp
.ne 2
.na
\fB\fBdladm modify-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-p\fR \fIpriority\fR] [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-delay\fR] [\fB-f\fR \fIforce-protocol\fR] [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR\fR
.ad

.sp .6
.RS 4n
Modify the operational parameters of an existing bridge. The options are the same as for the \fBcreate-bridge\fR subcommand, except that the \fB-l\fR option is not permitted. To add links to an existing bridge, use the \fBadd-bridge\fR subcommand.
.sp
Bridge parameter modification requires the \fBPRIV_SYS_DL_CONFIG\fR privilege.
.RE

.sp
.ne 2
.na
\fB\fBdladm delete-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fIbridge-name\fR\fR
.ad
.sp .6
.RS 4n
Delete a bridge instance. The bridge being deleted must not have any attached links. Use the \fBremove-bridge\fR subcommand to deactivate links before deleting a bridge.
.sp
Bridge deletion requires the \fBPRIV_SYS_DL_CONFIG\fR privilege.
.sp
The \fB-R\fR (\fB--root-dir\fR) option is the same as for the \fBcreate-bridge\fR subcommand.
.RE

.sp
.ne 2
.na
\fB\fBdladm add-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR\fR
.ad
.sp .6
.RS 4n
Add one or more links to an existing bridge. If multiple links are specified, and adding any one of them results in an error, the command fails and no changes are made to the system.
.sp
Link addition to a bridge requires the \fBPRIV_SYS_DL_CONFIG\fR privilege.
.sp
A link may be a member of at most one bridge. An error occurs when you attempt to add a link that already belongs to another bridge. To move a link from one bridge instance to another, remove it from the current bridge before adding it to a new one.
.sp
The links assigned to a bridge must not also be VLANs, VNICs, or tunnels. Only physical Ethernet datalinks, aggregation datalinks, wireless links, and Ethernet stubs are permitted to be assigned to a bridge.
.sp
Links assigned to a bridge must all have the same MTU. This is checked when the link is assigned. The link is added to the bridge in a deactivated form if it is not the first link on the bridge and it has a differing MTU.
.sp
Note that systems using bridging should not set the \fBeeprom\fR(1M) \fBlocal-mac-address?\fR variable to false.
.sp
The options are the same as for the \fBcreate-bridge\fR subcommand.
.RE

.sp
.ne 2
.na
\fB\fBdladm remove-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR\fR
.ad

```
.sp .6
.RS 4n
Remove one or more links from a bridge instance. If multiple links are
specified, and removing any one of them would result in an error, the command
fails and none are removed.
.sp
Link removal from a bridge requires the \fBPRIV_SYS_DL_CONFIG\fR privilege.
.sp
The options are the same as for the \fBcreate-bridge\fR subcommand.
.RE

.sp
.ne 2
.na
\fB\fBdladm show-bridge\fR [\fB-flt\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
[[\fB-p\fR] \fB-o\fR \fIfield\fR,...] [\fIbridge-name\fR]\fR
.ad
.sp .6
.RS 4n
Show the running status and configuration of bridges, their attached links,
learned forwarding entries, and \fBTRILL\fR nickname databases. When showing
overall bridge status and configuration, the bridge name can be omitted to show
all bridges. The other forms require a specified bridge.
.sp
The show-bridge subcommand accepts the following options:
.sp
.ne 2
.na
\fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
.ad
.sp .6
.RS 4n
Used with the \fB-s\fR option to specify an interval, in seconds, at which
statistics should be displayed. If this option is not specified, statistics
will be displayed only once.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR, \fB--statistics\fR\fR
.ad
.sp .6
.RS 4n
Display statistics for the specified bridges or for a given bridge’s attached
links. This option cannot be used with the \fB-f\fR and \fB-t\fR options.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Display using a stable machine-parsable format. See "Parsable Output Format,"
below.
.RE

.sp
.ne 2
.na
\fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
.ad
.sp .6
.RS 4n
```



```
A case-insensitive, comma-separated list of output fields to display. The field
names are described below. The special value all displays all fields. Each set
of fields has its own default set to display when \fB-o\fR is not specified.
.RE

By default, the \fBshow-bridge\fR subcommand shows bridge configuration. The
following fields can be shown:
.sp
.ne 2
.na
\fB\fBBRIDGE\fR\fR
.ad
.sp .6
.RS 4n
The name of the bridge.
.RE

.sp
.ne 2
.na
\fB\fBADDRESS\fR\fR
.ad
.sp .6
.RS 4n
The Bridge Unique Identifier value (MAC address).
.RE

.sp
.ne 2
.na
\fB\fBPRIORITY\fR\fR
.ad
.sp .6
.RS 4n
Configured priority value; set by \fB-p\fR with \fBcreate-bridge\fR and
\fBmodify-bridge\fR.
.RE

.sp
.ne 2
.na
\fB\fBMAXAGE\fR\fR
.ad
.sp .6
.RS 4n
Configured bridge maximum age; set by \fB-m\fR with \fBcreate-bridge\fR and
\fBmodify-bridge\fR.
.RE

.sp
.ne 2
.na
\fB\fBHELLOTIME\fR\fR
.ad
.sp .6
.RS 4n
Configured bridge hello time; set by \fB-h\fR with \fBcreate-bridge\fR and
\fBmodify-bridge\fR.
.RE

.sp
.ne 2
.na
\fB\fBFWDDELAY\fR\fR
.ad
.sp .6
```

```
.RS 4n
Configured forwarding delay; set by \fB-d\fR with \fBcreate-bridge\fR and
\fBmodify-bridge\fR.
.RE

.sp
.ne 2
.na
\fB\fBFORCEPROTO\fR\fR
.ad
.sp .6
.RS 4n
Configured forced maximum protocol; set by \fB-f\fR with \fBcreate-bridge\fR
and \fBmodify-bridge\fR.
.RE

.sp
.ne 2
.na
\fB\fBTCTIME\fR\fR
.ad
.sp .6
.RS 4n
Time, in seconds, since last topology change.
.RE

.sp
.ne 2
.na
\fB\fBTCCOUNT\fR\fR
.ad
.sp .6
.RS 4n
Count of the number of topology changes.
.RE

.sp
.ne 2
.na
\fB\fBTCHANGE\fR\fR
.ad
.sp .6
.RS 4n
This indicates that a topology change was detected.
.RE

.sp
.ne 2
.na
\fB\fBDESROOT\fR\fR
.ad
.sp .6
.RS 4n
Bridge Identifier of the root node.
.RE

.sp
.ne 2
.na
\fB\fBROOTCOST\fR\fR
.ad
.sp .6
.RS 4n
Cost of the path to the root node.
.RE
```



```
.sp
.ne 2
.na
\fB\fBROOTPORT\fR\fR
.ad
.sp .6
.RS 4n
Port number used to reach the root node.
.RE

.sp
.ne 2
.na
\fB\fBMAXAGE\fR\fR
.ad
.sp .6
.RS 4n
Maximum age value from the root node.
.RE

.sp
.ne 2
.na
\fB\fBHELLOTIME\fR\fR
.ad
.sp .6
.RS 4n
Hello time value from the root node.
.RE

.sp
.ne 2
.na
\fB\fBFWDDELAY\fR\fR
.ad
.sp .6
.RS 4n
Forward delay value from the root node.
.RE

.sp
.ne 2
.na
\fB\fBHOLDTIME\fR\fR
.ad
.sp .6
.RS 4n
Minimum BPDU interval.
.RE

By default, when the \fB-o\fR option is not specified, only the \fBBRIDGE\fR,
\fBADDRESS\fR, \fBPRIORITY\fR, and \fBDESROOT\fR fields are shown.
.sp
When the \fB-s\fR option is specified, the \fBshow-bridge\fR subcommand shows
bridge statistics. The following fields can be shown:
.sp
.ne 2
.na
\fB\fBBRIDGE\fR\fR
.ad
.sp .6
.RS 4n
Bridge name.
.RE

.sp
```

```
.ne 2
.na
\fB\fBDROPS\fR\fR
.ad
.sp .6
.RS 4n
Number of packets dropped due to resource problems.
.RE

.sp
.ne 2
.na
\fB\fBFORWARDS\fR\fR
.ad
.sp .6
.RS 4n
Number of packets forwarded from one link to another.
.RE

.sp
.ne 2
.na
\fB\fBMBCAST\fR\fR
.ad
.sp .6
.RS 4n
Number of multicast and broadcast packets handled by the bridge.
.RE

.sp
.ne 2
.na
\fB\fBRECV\fR\fR
.ad
.sp .6
.RS 4n
Number of packets received on all attached links.
.RE

.sp
.ne 2
.na
\fB\fBSENT\fR\fR
.ad
.sp .6
.RS 4n
Number of packets sent on all attached links.
.RE

.sp
.ne 2
.na
\fB\fBUNKNOWN\fR\fR
.ad
.sp .6
.RS 4n
Number of packets handled that have an unknown destination. Such packets are
sent to all links.
.RE

By default, when the \fB-o\fR option is not specified, only the \fBBRIDGE\fR,
\fBDROPS\fR, and \fBFORWARDS\fR fields are shown.
.sp
The \fBshow-bridge\fR subcommand also accepts the following options:
.sp
.ne 2
```



```
.na
\fB\fB-l\fR, \fB--link\fR\fR
.ad
.sp .6
.RS 4n
Displays link-related status and statistics information for all links attached
to a single bridge instance. By using this option and without the \fB-s\fR
option, the following fields can be displayed for each link:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The link name.
.RE

.sp
.ne 2
.na
\fB\fBINDEX\fR\fR
.ad
.sp .6
.RS 4n
Port (link) index number on the bridge.
.RE

.sp
.ne 2
.na
\fB\fBSTATE\fR\fR
.ad
.sp .6
.RS 4n
State of the link. The state can be \fBdisabled\fR, \fBdiscarding\fR,
\fBlearning\fR, \fBforwarding\fR, \fBnon-stp\fR, or \fBbad-mtu\fR.
.RE

.sp
.ne 2
.na
\fB\fBUPTIME\fR\fR
.ad
.sp .6
.RS 4n
Number of seconds since the last reset or initialization.
.RE

.sp
.ne 2
.na
\fB\fBOPERCOST\fR\fR
.ad
.sp .6
.RS 4n
Actual cost in use (1-65535).
.RE

.sp
.ne 2
.na
\fB\fBOPERP2P\fR\fR
.ad
.sp .6
.RS 4n
```

```
2104 This indicates whether point-to-point (\fBP2P\fR) mode been detected.
2105 .RE

2107 .sp
2108 .ne 2
2109 .na
2110 \fB\fBOPEREDGE\fR\fR
2111 .ad
2112 .sp .6
2113 .RS 4n
2114 This indicates whether edge mode has been detected.
2115 .RE

2117 .sp
2118 .ne 2
2119 .na
2120 \fB\fBDESROOT\fR\fR
2121 .ad
2122 .sp .6
2123 .RS 4n
2124 The Root Bridge Identifier that has been seen on this port.
2125 .RE

2127 .sp
2128 .ne 2
2129 .na
2130 \fB\fBDESCOST\fR\fR
2131 .ad
2132 .sp .6
2133 .RS 4n
2134 Path cost to the network root node through the designated port.
2135 .RE

2137 .sp
2138 .ne 2
2139 .na
2140 \fB\fBDESBRIDGE\fR\fR
2141 .ad
2142 .sp .6
2143 .RS 4n
2144 Bridge Identifier for this port.
2145 .RE

2147 .sp
2148 .ne 2
2149 .na
2150 \fB\fBDESPORT\fR\fR
2151 .ad
2152 .sp .6
2153 .RS 4n
2154 The ID and priority of the port used to transmit configuration messages for
2155 this port.
2156 .RE

2158 .sp
2159 .ne 2
2160 .na
2161 \fB\fBTCACK\fR\fR
2162 .ad
2163 .sp .6
2164 .RS 4n
2165 This indicates whether Topology Change Acknowledge has been seen.
2166 .RE

2168 When the \fB-l\fR option is specified without the \fB-o\fR option, only the
2169 \fBLINK\fR, \fBSTATE\fR, \fBUPTIME\fR, and \fBDESROOT\fR fields are shown.
```



```
2170 .sp
2171 When the \fB-l\fR option is specified, the \fB-s\fR option can be used to
2172 display the following fields for each link:
2173 .sp
2174 .ne 2
2175 .na
2176 \fB\fBLINK\fR\fR
2177 .ad
2178 .sp .6
2179 .RS 4n
2180 Link name.
2181 .RE

2183 .sp
2184 .ne 2
2185 .na
2186 \fB\fBCFGBPDU\fR\fR
2187 .ad
2188 .sp .6
2189 .RS 4n
2190 Number of configuration BPDUs received.
2191 .RE

2193 .sp
2194 .ne 2
2195 .na
2196 \fB\fBTCNBPDU\fR\fR
2197 .ad
2198 .sp .6
2199 .RS 4n
2200 Number of topology change BPDUs received.
2201 .RE

2203 .sp
2204 .ne 2
2205 .na
2206 \fB\fBRSTPBPDU\fR\fR
2207 .ad
2208 .sp .6
2209 .RS 4n
2210 Number of Rapid Spanning Tree BPDUs received.
2211 .RE

2213 .sp
2214 .ne 2
2215 .na
2216 \fB\fBTXBPDU\fR\fR
2217 .ad
2218 .sp .6
2219 .RS 4n
2220 Number of BPDUs transmitted.
2221 .RE

2223 .sp
2224 .ne 2
2225 .na
2226 \fB\fBDROPS\fR\fR
2227 .ad
2228 .sp .6
2229 .RS 4n
2230 Number of packets dropped due to resource problems.
2231 .RE

2233 .sp
2234 .ne 2
2235 .na
```

```
2236 \fB\fBRECV\fR\fR
2237 .ad
2238 .sp .6
2239 .RS 4n
2240 Number of packets received by the bridge.
2241 .RE

2243 .sp
2244 .ne 2
2245 .na
2246 \fB\fBXMIT\fR\fR
2247 .ad
2248 .sp .6
2249 .RS 4n
2250 Number of packets sent by the bridge.
2251 .RE

2253 When the \fB-o\fR option is not specified, only the \fBLINK\fR, \fBDROPS\fR,
2254 \fBRECV\fR, and \fBXMIT\fR fields are shown.
2255 .RE

2257 .sp
2258 .ne 2
2259 .na
2260 \fB\fB-f\fR, \fB--forwarding\fR\fR
2261 .ad
2262 .sp .6
2263 .RS 4n
2264 Displays forwarding entries for a single bridge instance. With this option, the
2265 following fields can be shown for each forwarding entry:
2266 .sp
2267 .ne 2
2268 .na
2269 \fB\fBDEST\fR\fR
2270 .ad
2271 .sp .6
2272 .RS 4n
2273 Destination MAC address.
2274 .RE

2276 .sp
2277 .ne 2
2278 .na
2279 \fB\fBAGE\fR\fR
2280 .ad
2281 .sp .6
2282 .RS 4n
2283 Age of entry in seconds and milliseconds. Omitted for local entries.
2284 .RE

2286 .sp
2287 .ne 2
2288 .na
2289 \fB\fBFLAGS\fR\fR
2290 .ad
2291 .sp .6
2292 .RS 4n
2293 The \fBL\fR (local) flag is shown if the MAC address belongs to an attached
2294 link or to a VNIC on one of the attached links.
2295 .RE

2297 .sp
2298 .ne 2
2299 .na
2300 \fB\fBOUTPUT\fR\fR
2301 .ad
```



```
2302 .sp .6
2303 .RS 4n
2304 For local entries, this is the name of the attached link that has the MAC
2305 address. Otherwise, for bridges that use Spanning Tree Protocol, this is the
2306 output interface name. For RBridges, this is the output \fBTRILL\fR nickname.
2307 .RE

2309 When the \fB-o\fR option is not specified, the \fBDEST\fR, \fBAGE\fR,
2310 \fBFLAGS\fR, and \fBOUTPUT\fR fields are shown.
2311 .RE

2313 .sp
2314 .ne 2
2315 .na
2316 \fB\fB-t\fR, \fB--trill\fR\fR
2317 .ad
2318 .sp .6
2319 .RS 4n
2320 Displays \fBTRILL\fR nickname entries for a single bridge instance. With this
2321 option, the following fields can be shown for each \fBTRILL\fR nickname entry:
2322 .sp
2323 .ne 2
2324 .na
2325 \fB\fBNICK\fR\fR
2326 .ad
2327 .sp .6
2328 .RS 4n
2329 \fBTRILL\fR nickname for this RBridge, which is a number from 1 to 65535.
2330 .RE

2332 .sp
2333 .ne 2
2334 .na
2335 \fB\fBFLAGS\fR\fR
2336 .ad
2337 .sp .6
2338 .RS 4n
2339 The \fBL\fR flag is shown if the nickname identifies the local system.
2340 .RE

2342 .sp
2343 .ne 2
2344 .na
2345 \fB\fBLINK\fR\fR
2346 .ad
2347 .sp .6
2348 .RS 4n
2349 Link name for output when sending messages to this RBridge.
2350 .RE

2352 .sp
2353 .ne 2
2354 .na
2355 \fB\fBNEXTHOP\fR\fR
2356 .ad
2357 .sp .6
2358 .RS 4n
2359 MAC address of the next hop RBridge that is used to reach the RBridge with this
2360 nickname.
2361 .RE

2363 When the \fB-o\fR option is not specified, the \fBNICK\fR, \fBFLAGS\fR,
2364 \fBLINK\fR, and \fBNEXTHOP\fR fields are shown.
2365 .RE

2367 .RE
```

```
2369 .sp
2370 .ne 2
2371 .na
2372 \fB\fBdladm create-vlan\fR [\fB-ft\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
2373 \fIether-link\fR \fB-v\fR \fIvid\fR [\fIvlan-link\fR]\fR
2374 .ad
2375 .sp .6
2376 .RS 4n
2377 Create a tagged VLAN link with an ID of \fIvid\fR over Ethernet link
2378 \fIether-link\fR. The name of the VLAN link can be specified as
2379 \fIvlan\fR-\fIlink\fR. If the name is not specified, a name will be
2380 automatically generated (assuming that \fIether-link\fR is \fIname\fR\fIPPA\fR)
2381 as:
2382 .sp
2383 .in +2
2384 .nf
2385 <\fIname\fR><1000 * \fIvlan-tag\fR + \fIPPA\fR>
2386 .fi
2387 .in -2
2388 .sp

2390 For example, if \fIether-link\fR is \fBbge1\fR and \fIvid\fR is 2, the name
2391 generated is \fBbge2001\fR.
2392 .sp
2393 .ne 2
2394 .na
2395 \fB\fB-f\fR, \fB--force\fR\fR
2396 .ad
2397 .sp .6
2398 .RS 4n
2399 Force the creation of the VLAN link. Some devices do not allow frame sizes
2400 large enough to include a VLAN header. When creating a VLAN link over such a
2401 device, the \fB-f\fR option is needed, and the MTU of the IP interfaces on the
2402 resulting VLAN must be set to 1496 instead of 1500.
2403 .RE

2405 .sp
2406 .ne 2
2407 .na
2408 \fB\fB-l\fR \fIether-link\fR\fR
2409 .ad
2410 .sp .6
2411 .RS 4n
2412 Specifies Ethernet link over which VLAN is created.
2413 .RE

2415 .sp
2416 .ne 2
2417 .na
2418 \fB\fB-t\fR, \fB--temporary\fR\fR
2419 .ad
2420 .sp .6
2421 .RS 4n
2422 Specifies that the VLAN link is temporary. Temporary VLAN links last until the
2423 next reboot.
2424 .RE

2426 .sp
2427 .ne 2
2428 .na
2429 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
2430 .ad
2431 .sp .6
2432 .RS 4n
2433 See "Options," above.
```



```
2434 .RE

2436 .RE

2438 .sp
2439 .ne 2
2440 .na
2441 \fB\fBdladm delete-vlan\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
2442 \fIvlan-link\fR\fR
2443 .ad
2444 .sp .6
2445 .RS 4n
2446 Delete the VLAN link specified.
2447 .sp
2448 The \fBdelete-vlan\fRsubcommand accepts the following options:
2449 .sp
2450 .ne 2
2451 .na
2452 \fB\fB-t\fR, \fB--temporary\fR\fR
2453 .ad
2454 .sp .6
2455 .RS 4n
2456 Specifies that the deletion is temporary. Temporary deletions last until the
2457 next reboot.
2458 .RE

2460 .sp
2461 .ne 2
2462 .na
2463 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
2464 .ad
2465 .sp .6
2466 .RS 4n
2467 See "Options," above.
2468 .RE

2470 .RE

2472 .sp
2473 .ne 2
2474 .na
2475 \fB\fBdladm show-vlan\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
2476 [\fIvlan-link\fR]\fR
2477 .ad
2478 .sp .6
2479 .RS 4n
2480 Display VLAN configuration for all VLAN links or for the specified VLAN link.
2481 .sp
2482 The \fBshow-vlan\fRsubcommand accepts the following options:
2483 .sp
2484 .ne 2
2485 .na
2486 \fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
2487 .ad
2488 .sp .6
2489 .RS 4n
2490 A case-insensitive, comma-separated list of output fields to display. The field
2491 name must be one of the fields listed below, or the special value \fBall\fR, to
2492 display all fields. For each VLAN link, the following fields can be displayed:
2493 .sp
2494 .ne 2
2495 .na
2496 \fB\fBLINK\fR\fR
2497 .ad
2498 .sp .6
2499 .RS 4n
```

```
2500 The name of the VLAN link.
2501 .RE

2503 .sp
2504 .ne 2
2505 .na
2506 \fB\fBVID\fR\fR
2507 .ad
2508 .sp .6
2509 .RS 4n
2510 The ID associated with the VLAN.
2511 .RE

2513 .sp
2514 .ne 2
2515 .na
2516 \fB\fBOVER\fR\fR
2517 .ad
2518 .sp .6
2519 .RS 4n
2520 The name of the physical link over which this VLAN is configured.
2521 .RE

2523 .sp
2524 .ne 2
2525 .na
2526 \fB\fBFLAGS\fR\fR
2527 .ad
2528 .sp .6
2529 .RS 4n
2530 A set of flags associated with the VLAN link. Possible flags are:
2531 .sp
2532 .ne 2
2533 .na
2534 \fB\fBf\fR\fR
2535 .ad
2536 .sp .6
2537 .RS 4n
2538 The VLAN was created using the \fB-f\fR option to \fBcreate-vlan\fR.
2539 .RE

2541 .sp
2542 .ne 2
2543 .na
2544 \fB\fBi\fR\fR
2545 .ad
2546 .sp .6
2547 .RS 4n
2548 The VLAN was implicitly created when the DLPI link was opened. These VLAN links
2549 are automatically deleted on last close of the DLPI link (for example, when the
2550 IP interface associated with the VLAN link is unplumbed).
2551 .RE

2553 Additional flags might be defined in the future.
2554 .RE

2556 .RE

2558 .sp
2559 .ne 2
2560 .na
2561 \fB\fB-p\fR, \fB--parseable\fR\fR
2562 .ad
2563 .sp .6
2564 .RS 4n
2565 Display using a stable machine-parseable format. The \fB-o\fR option is
```



```
2566 required with \fB-p\fR. See "Parseable Output Format", below.
2567 .RE

2569 .sp
2570 .ne 2
2571 .na
2572 \fB\fB-P\fR, \fB--persistent\fR\fR
2573 .ad
2574 .sp .6
2575 .RS 4n
2576 Display the persistent VLAN configuration rather than the state of the running
2577 system.
2578 .RE

2580 .RE

2582 .sp
2583 .ne 2
2584 .na
2585 \fB\fBdladm scan-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
2586 [\fIwifi-link\fR]\fR
2587 .ad
2588 .sp .6
2589 .RS 4n
2590 Scans for \fBWiFi\fR networks, either on all \fBWiFi\fR links, or just on the
2591 specified \fIwifi-link\fR.
2592 .sp
2593 By default, currently all fields but \fBBSSTYPE\fR are displayed.
2594 .sp
2595 .ne 2
2596 .na
2597 \fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
2598 .ad
2599 .sp .6
2600 .RS 4n
2601 A case-insensitive, comma-separated list of output fields to display. The field
2602 name must be one of the fields listed below, or the special value \fBall\fR to
2603 display all fields. For each \fBWiFi\fR network found, the following fields can
2604 be displayed:
2605 .sp
2606 .ne 2
2607 .na
2608 \fB\fBLINK\fR\fR
2609 .ad
2610 .sp .6
2611 .RS 4n
2612 The name of the link the \fBWiFi\fR network is on.
2613 .RE

2615 .sp
2616 .ne 2
2617 .na
2618 \fB\fBESSID\fR\fR
2619 .ad
2620 .sp .6
2621 .RS 4n
2622 The \fBESSID\fR (name) of the \fBWiFi\fR network.
2623 .RE

2625 .sp
2626 .ne 2
2627 .na
2628 \fB\fBBSSID\fR\fR
2629 .ad
2630 .sp .6
2631 .RS 4n
```

Either the hardware address of the \fBWiFi\fR network's Access Point (for \fBBSS\fR networks), or the \fBWiFi\fR network's randomly generated unique token (for \fBIBSS\fR networks).
.RE

.sp
.ne 2
.na
\fB\fBSEC\fR\fR
.ad
.sp .6
.RS 4n
Either \fBnone\fR for a \fBWiFi\fR network that uses no security, \fBwep\fR for a \fBWiFi\fR network that requires WEP (Wired Equivalent Privacy), or \fBwpa\fR for a WiFi network that requires WPA (Wi-Fi Protected Access).
.RE

.sp
.ne 2
.na
\fB\fBMODE\fR\fR
.ad
.sp .6
.RS 4n
The supported connection modes: one or more of \fBa\fR, \fBb\fR, or \fBg\fR.
.RE

.sp
.ne 2
.na
\fB\fBSTRENGTH\fR\fR
.ad
.sp .6
.RS 4n
The strength of the signal: one of \fBexcellent\fR, \fBvery good\fR, \fBgood\fR, \fBweak\fR, or \fBvery weak\fR.
.RE

.sp
.ne 2
.na
\fB\fBSPEED\fR\fR
.ad
.sp .6
.RS 4n
The maximum speed of the \fBWiFi\fR network, in megabits per second.
.RE

.sp
.ne 2
.na
\fB\fBBSSTYPE\fR\fR
.ad
.sp .6
.RS 4n
Either \fBbss\fR for \fBBSS\fR (infrastructure) networks, or \fBibss\fR for \fBIBSS\fR (ad-hoc) networks.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-p\fR, \fB--parseable\fR\fR
.ad

.sp .6
.RS 4n
Display using a stable machine-parseable format. The \fB-o\fR option is required with \fB-p\fR. See "Parseable Output Format", below.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm connect-wifi\fR [\fB-e\fR \fIessid\fR] [\fB-i\fR \fIbssid\fR] [\fB-k\fR \fIkey\fR,...] [\fB-s\fR \fBnone\fR | \fBwep\fR | \fBwpa\fR] [\fB-a\fR \fBopen\fR|\fBshared\fR] [\fB-b\fR \fBbss\fR|\fBibss\fR] [\fB-c\fR] [\fB-m\fR \fBa\fR|\fBb\fR|\fBg\fR] [\fB-T\fR \fItime\fR] [\fIwifi-link\fR]\fR
.ad
.sp .6
.RS 4n
Connects to a \fBWiFi\fR network. This consists of four steps: \fIdiscovery\fR, \fIfiltration\fR, \fIprioritization\fR, and \fIassociation\fR. However, to enable connections to non-broadcast \fBWiFi\fR networks and to improve performance, if a \fBBSSID\fR or \fBESSID\fR is specified using the \fB-e\fR or \fB-i\fR options, then the first three steps are skipped and \fBconnect-wifi\fR immediately attempts to associate with a \fBBSSID\fR or \fBESSID\fR that matches the rest of the provided parameters. If this association fails, but there is a possibility that other networks matching the specified criteria exist, then the traditional discovery process begins as specified below.
.sp
The discovery step finds all available \fBWiFi\fR networks on the specified WiFi link, which must not yet be connected. For administrative convenience, if there is only one \fBWiFi\fR link on the system, \fIwifi-link\fR can be omitted.
.sp
Once discovery is complete, the list of networks is filtered according to the value of the following options:
.sp
.ne 2
.na
\fB\fB-e\fR \fIessid,\fR \fB--essid\fR=\fIessid\fR\fR
.ad
.sp .6
.RS 4n
Networks that do not have the same \fIessid\fR are filtered out.
.RE

.sp
.ne 2
.na
\fB\fB-b\fR \fBbss\fR|\fBibss\fR, \fB--bsstype\fR=\fBbss\fR|\fBibss\fR\fR
.ad
.sp .6
.RS 4n
Networks that do not have the same \fBbsstype\fR are filtered out.
.RE

.sp
.ne 2
.na
\fB\fB-m\fR \fBa\fR|\fBb\fR|\fBg\fR, \fB--mode\fR=\fBa\fR|\fBb\fR|\fBg\fR\fR
.ad
.sp .6
.RS 4n
Networks not appropriate for the specified 802.11 mode are filtered out.
.RE

.sp

```
.ne 2
.na
\fB\fB-k\fR \fIkey,...\fR, \fB--key\fR=\fIkey, ...\fR\fR
.ad
.sp .6
.RS 4n
Use the specified \fBsecobj\fR named by the key to connect to the network.
Networks not appropriate for the specified keys are filtered out.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR \fBnone\fR|\fBwep\fR|\fBwpa\fR,
\fB--sec\fR=\fBnone\fR|\fBwep\fR|\fBwpa\fR\fR
.ad
.sp .6
.RS 4n
Networks not appropriate for the specified security mode are filtered out.
.RE

Next, the remaining networks are prioritized, first by signal strength, and
then by maximum speed. Finally, an attempt is made to associate with each
network in the list, in order, until one succeeds or no networks remain.
.sp
In addition to the options described above, the following options also control
the behavior of \fBconnect-wifi\fR:
.sp
.ne 2
.na
\fB\fB-a\fR \fBopen\fR|\fBshared\fR, \fB--auth\fR=\fBopen\fR|\fBshared\fR\fR
.ad
.sp .6
.RS 4n
Connect using the specified authentication mode. By default, \fBopen\fR and
\fBshared\fR are tried in order.
.RE

.sp
.ne 2
.na
\fB\fB-c\fR, \fB--create-ibss\fR\fR
.ad
.sp .6
.RS 4n
Used with \fB-b ibss\fR to create a new ad-hoc network if one matching the
specified \fBESSID\fR cannot be found. If no \fBESSID\fR is specified, then
\fB-c -b ibss\fR always triggers the creation of a new ad-hoc network.
.RE

.sp
.ne 2
.na
\fB\fB-T\fR \fItime\fR, \fB--timeout\fR=\fItime\fR\fR
.ad
.sp .6
.RS 4n
Specifies the number of seconds to wait for association to succeed. If
\fItime\fR is \fBforever\fR, then the associate will wait indefinitely. The
current default is ten seconds, but this might change in the future. Timeouts
shorter than the default might not succeed reliably.
.RE

.sp
.ne 2
.na
```



```
\fB\fB-k\fR \fIkey,...\fR, \fB--key\fR=\fIkey,...\fR\fR
.ad
.sp .6
.RS 4n
In addition to the filtering previously described, the specified keys will be
used to secure the association. The security mode to use will be based on the
key class; if a security mode was explicitly specified, it must be compatible
with the key class. All keys must be of the same class.
.sp
For security modes that support multiple key slots, the slot to place the key
will be specified by a colon followed by an index. Therefore, \fB-k mykey:3\fR
places \fBmykey\fR in slot 3. By default, slot 1 is assumed. For security modes
that support multiple keys, a comma-separated list can be specified, with the
first key being the active key.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm disconnect-wifi\fR [\fB-a\fR] [\fIwifi-link\fR]\fR
.ad
.sp .6
.RS 4n
Disconnect from one or more \fBWiFi\fR networks. If \fIwifi-link\fR specifies a
connected \fBWiFi\fR link, then it is disconnected. For administrative
convenience, if only one \fBWiFi\fR link is connected, \fIwifi-link\fR can be
omitted.
.sp
.ne 2
.na
\fB\fB-a\fR, \fB--all-links\fR\fR
.ad
.sp .6
.RS 4n
Disconnects from all connected links. This is primarily intended for use by
scripts.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm show-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR,...]
[\fIwifi-link\fR]\fR
.ad
.sp .6
.RS 4n
Shows \fBWiFi\fR configuration information either for all \fBWiFi\fR links or
for the specified link \fIwifi-link\fR.
.sp
.ne 2
.na
\fB\fB-o\fR \fIfield,...\fR, \fB--output\fR=\fIfield\fR\fR
.ad
.sp .6
.RS 4n
A case-insensitive, comma-separated list of output fields to display. The field
name must be one of the fields listed below, or the special value \fBall\fR, to
display all fields. For each \fBWiFi\fR link, the following fields can be
displayed:
.sp
.ne 2
.na
```

```
2896 \fB\fBLINK\fR\fR
2897 .ad
2898 .sp .6
2899 .RS 4n
2900 The name of the link being displayed.
2901 .RE

2903 .sp
2904 .ne 2
2905 .na
2906 \fB\fBSTATUS\fR\fR
2907 .ad
2908 .sp .6
2909 .RS 4n
2910 Either \fBconnected\fR if the link is connected, or \fBdisconnected\fR if it is
2911 not connected. If the link is disconnected, all remaining fields have the value
2912 \fB--\fR.
2913 .RE

2915 .sp
2916 .ne 2
2917 .na
2918 \fB\fBESSID\fR\fR
2919 .ad
2920 .sp .6
2921 .RS 4n
2922 The \fBESSID\fR (name) of the connected \fBWiFi\fR network.
2923 .RE

2925 .sp
2926 .ne 2
2927 .na
2928 \fB\fBBSSID\fR\fR
2929 .ad
2930 .sp .6
2931 .RS 4n
2932 Either the hardware address of the \fBWiFi\fR network's Access Point (for
2933 \fBBSS\fR networks), or the \fBWiFi\fR network's randomly generated unique
2934 token (for \fBIBSS\fR networks).
2935 .RE

2937 .sp
2938 .ne 2
2939 .na
2940 \fB\fBSEC\fR\fR
2941 .ad
2942 .sp .6
2943 .RS 4n
2944 Either \fBnone\fR for a \fBWiFi\fR network that uses no security, \fBwep\fR for
2945 a \fBWiFi\fR network that requires WEP, or \fBwpa\fR for a WiFi network that
2946 requires WPA.
2947 .RE

2949 .sp
2950 .ne 2
2951 .na
2952 \fB\fBMODE\fR\fR
2953 .ad
2954 .sp .6
2955 .RS 4n
2956 The supported connection modes: one or more of \fBa\fR, \fBb\fR, or \fBg\fR.
2957 .RE

2959 .sp
2960 .ne 2
2961 .na
```



```
2962 \fB\fBSTRENGTH\fR\fR
2963 .ad
2964 .sp .6
2965 .RS 4n
2966 The connection strength: one of \fBexcellent\fR, \fBvery good\fR, \fBgood\fR,
2967 \fBweak\fR, or \fBvery weak\fR.
2968 .RE

2970 .sp
2971 .ne 2
2972 .na
2973 \fB\fBSPEED\fR\fR
2974 .ad
2975 .sp .6
2976 .RS 4n
2977 The connection speed, in megabits per second.
2978 .RE

2980 .sp
2981 .ne 2
2982 .na
2983 \fB\fBAUTH\fR\fR
2984 .ad
2985 .sp .6
2986 .RS 4n
2987 Either \fBopen\fR or \fBshared\fR (see \fBconnect-wifi\fR).
2988 .RE

2990 .sp
2991 .ne 2
2992 .na
2993 \fB\fBBSSTYPE\fR\fR
2994 .ad
2995 .sp .6
2996 .RS 4n
2997 Either \fBbss\fR for \fBBSS\fR (infrastructure) networks, or \fBibss\fR for
2998 \fBIBSS\fR (ad-hoc) networks.
2999 .RE

3001 By default, currently all fields but \fBAUTH\fR, \fBBSSID\fR, \fBBSSTYPE\fR are
3002 displayed.
3003 .RE

3005 .sp
3006 .ne 2
3007 .na
3008 \fB\fB-p\fR, \fB--parseable\fR\fR
3009 .ad
3010 .sp .6
3011 .RS 4n
3012 Displays using a stable machine-parseable format. The \fB-o\fR option is
3013 required with \fB-p\fR. See "Parseable Output Format", below.
3014 .RE

3016 .RE

3018 .sp
3019 .ne 2
3020 .na
3021 \fB\fBdladm show-ether\fR [\fB-x\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR,...]
3022 [\fIether-link\fR]\fR
3023 .ad
3024 .sp .6
3025 .RS 4n
3026 Shows state information either for all physical Ethernet links or for a
3027 specified physical Ethernet link.
```

```
3028 .sp
3029 The \fBshow-ether\fR subcommand accepts the following options:
3030 .sp
3031 .ne 2
3032 .na
3033 \fB\fB-o\fR \fIfield\fR,..., \fB--output\fR=\fIfield\fR\fR
3034 .ad
3035 .sp .6
3036 .RS 4n
3037 A case-insensitive, comma-separated list of output fields to display. The field
3038 name must be one of the fields listed below, or the special value \fBall\fR to
3039 display all fields. For each link, the following fields can be displayed:
3040 .sp
3041 .ne 2
3042 .na
3043 \fB\fBLINK\fR\fR
3044 .ad
3045 .sp .6
3046 .RS 4n
3047 The name of the link being displayed.
3048 .RE

3050 .sp
3051 .ne 2
3052 .na
3053 \fB\fBPTYPE\fR\fR
3054 .ad
3055 .sp .6
3056 .RS 4n
3057 Parameter type, where \fBcurrent\fR indicates the negotiated state of the link,
3058 \fBcapable\fR indicates capabilities supported by the device, \fBadv\fR
3059 indicates the advertised capabilities, and \fBpeeradv\fR indicates the
3060 capabilities advertised by the link-partner.
3061 .RE

3063 .sp
3064 .ne 2
3065 .na
3066 \fB\fBSTATE\fR\fR
3067 .ad
3068 .sp .6
3069 .RS 4n
3070 The state of the link.
3071 .RE

3073 .sp
3074 .ne 2
3075 .na
3076 \fB\fBAUTO\fR\fR
3077 .ad
3078 .sp .6
3079 .RS 4n
3080 A \fByes\fR/\fBno\fR value indicating whether auto-negotiation is advertised.
3081 .RE

3083 .sp
3084 .ne 2
3085 .na
3086 \fB\fBSPEED-DUPLEX\fR\fR
3087 .ad
3088 .sp .6
3089 .RS 4n
3090 Combinations of speed and duplex values available. The units of speed are
3091 encoded with a trailing suffix of \fBG\fR (Gigabits/s) or \fBM\fR (Mb/s).
3092 Duplex values are encoded as \fBf\fR (full-duplex) or \fBh\fR (half-duplex).
3093 .RE
```



```
3095 .sp
3096 .ne 2
3097 .na
3098 \fB\fBPAUSE\fR\fR
3099 .ad
3100 .sp .6
3101 .RS 4n
3102 Flow control information. Can be \fBno\fR, indicating no flow control is
3103 available; \fBtx\fR, indicating that the end-point can transmit pause frames,
3104 but ignores any received pause frames; \fBrx\fR, indicating that the end-point
3105 receives and acts upon received pause frames; or \fBbi\fR, indicating
3106 bi-directional flow-control.
3107 .RE

3109 .sp
3110 .ne 2
3111 .na
3112 \fB\fBREM_FAULT\fR\fR
3113 .ad
3114 .sp .6
3115 .RS 4n
3116 Fault detection information. Valid values are \fBnone\fR or \fBfault\fR.
3117 .RE

3119 By default, all fields except \fBREM_FAULT\fR are displayed for the "current"
3120 \fBPTYPE\fR.
3121 .RE

3123 .sp
3124 .ne 2
3125 .na
3126 \fB\fB-p\fR, \fB--parseable\fR\fR
3127 .ad
3128 .sp .6
3129 .RS 4n
3130 Displays using a stable machine-parseable format. The \fB-o\fR option is
3131 required with \fB-p\fR. See "Parseable Output Format", below.
3132 .RE

3134 .sp
3135 .ne 2
3136 .na
3137 \fB\fB-x\fR, \fB--extended\fR\fR
3138 .ad
3139 .sp .6
3140 .RS 4n
3141 Extended output is displayed for \fBPTYPE\fR values of \fBcurrent\fR,
3142 \fBcapable\fR, \fBadv\fR and \fBpeeradv\fR.
3143 .RE

3145 .RE

3147 .sp
3148 .ne 2
3149 .na
3150 \fB\fBdladm set-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-p\fR
3151 \fIprop\fR=\fIvalue\fR[,...] \fIlink\fR\fR
3152 .ad
3153 .sp .6
3154 .RS 4n
3155 Sets the values of one or more properties on the link specified. The list of
3156 properties and their possible values depend on the link type, the network
3157 device driver, and networking hardware. These properties can be retrieved using
3158 \fBshow-linkprop\fR.
3159 .sp
```

```
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the changes are temporary. Temporary changes last until the next
reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIprop\fR=\fIvalue\fR[,...], \fB--prop\fR
\fIprop\fR=\fIvalue\fR[,...]\fR
.ad
.br
.na
\fB\fR
.ad
.sp .6
.RS 4n
A comma-separated list of properties to set to the specified values.
.RE

Note that when the persistent value is set, the temporary value changes to the
same value.
.RE

.sp
.ne 2
.na
\fB\fBdladm reset-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-p\fR
\fIprop\fR,...] \fIlink\fR\fR
.ad
.sp .6
.RS 4n
Resets one or more properties to their values on the link specified. Properties
are reset to the values they had at startup. If no properties are specified,
all properties are reset. See \fBshow-linkprop\fR for a description of
properties.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the resets are temporary. Values are reset to default values.
Temporary resets last until the next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
```



```
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIprop, ...\fR, \fB--prop\fR=\fIprop, ...\fR\fR
.ad
.sp .6
.RS 4n
A comma-separated list of properties to reset.
.RE

Note that when the persistent value is reset, the temporary value changes to
the same value.
.RE

.sp
.ne 2
.na
\fB\fBdladm show-linkprop\fR [\fB-P\fR] [[\fB-c\fR] \fB-o\fR
\fIfield\fR[,...]][\fB-p\fR \fIprop\fR[,...]] [\fIlink\fR]\fR
.ad
.sp .6
.RS 4n
Show the current or persistent values of one or more properties, either for all
datalinks or for the specified link. By default, current values are shown. If
no properties are specified, all available link properties are displayed. For
each property, the following fields are displayed:
.sp
.ne 2
.na
\fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR\fR
.ad
.sp .6
.RS 4n
A case-insensitive, comma-separated list of output fields to display. The field
name must be one of the fields listed below, or the special value \fBall\fR to
display all fields. For each link, the following fields can be displayed:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the datalink.
.RE

.sp
.ne 2
.na
\fB\fBPROPERTY\fR\fR
.ad
.sp .6
.RS 4n
The name of the property.
.RE

.sp
.ne 2
.na
\fB\fBPERM\fR\fR
```

```
.ad
.sp .6
.RS 4n
The read/write permissions of the property. The value shown is one of \fBro\fR
or \fBrw\fR.
.RE

.sp
.ne 2
.na
\fB\fBVALUE\fR\fR
.ad
.sp .6
.RS 4n
The current (or persistent) property value. If the value is not set, it is
shown as \fB--\fR. If it is unknown, the value is shown as \fB?\fR. Persistent
values that are not set or have been reset will be shown as \fB--\fR and will
use the system \fBDEFAULT\fR value (if any).
.RE

.sp
.ne 2
.na
\fB\fBDEFAULT\fR\fR
.ad
.sp .6
.RS 4n
The default value of the property. If the property has no default value,
\fB--\fR is shown.
.RE

.sp
.ne 2
.na
\fB\fBPOSSIBLE\fR\fR
.ad
.sp .6
.RS 4n
A comma-separated list of the values the property can have. If the values span
a numeric range, \fImin\fR - \fImax\fR might be shown as shorthand. If the
possible values are unknown or unbounded, \fB--\fR is shown.
.RE

The list of properties depends on the link type and network device driver, and
the available values for a given property further depends on the underlying
network hardware and its state. General link properties are documented in the
\fBLINK PROPERTIES\fR section. However, link properties that begin with
"\fB_\fR" (underbar) are specific to a given link or its underlying network
device and subject to change or removal. See the appropriate network device
driver man page for details.
.RE

.sp
.ne 2
.na
\fB\fB-c\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Display using a stable machine-parseable format. The \fB-o\fR option is
required with this option. See "Parseable Output Format", below.
.RE

.sp
.ne 2
.na
```



```
\fB\fB-P\fR, \fB--persistent\fR\fR
.ad
.sp .6
.RS 4n
Display persistent link property information
.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIprop, ...\fR, \fB--prop\fR=\fIprop, ...\fR\fR
.ad
.sp .6
.RS 4n
A comma-separated list of properties to show. See the sections on link
properties following subcommand descriptions.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm create-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-f\fR
\fIfile\fR] \fB-c\fR \fIclass\fR \fIsecobj\fR\fR
.ad
.sp .6
.RS 4n
Create a secure object named \fIsecobj\fR in the specified \fIclass\fR to be
later used as a WEP or WPA key in connecting to an encrypted network. The value
of the secure object can either be provided interactively or read from a file.
The sequence of interactive prompts and the file format depends on the class of
the secure object.
.sp
Currently, the classes \fBwep\fR and \fBwpa\fR are supported. The \fBWEP\fR
(Wired Equivalent Privacy) key can be either 5 or 13 bytes long. It can be
provided either as an \fBASCII\fR or hexadecimal string -- thus, \fB12345\fR
and \fB0x3132333435\fR are equivalent 5-byte keys (the \fB0x\fR prefix can be
omitted). A file containing a \fBWEP\fR key must consist of a single line using
either \fBWEP\fR key format. The WPA (Wi-Fi Protected Access) key must be
provided as an ASCII string with a length between 8 and 63 bytes.
.sp
This subcommand is only usable by users or roles that belong to the "Network
Link Security" \fBRBAC\fR profile.
.sp
.ne 2
.na
\fB\fB-c\fR \fIclass\fR, \fB--class\fR=\fIclass\fR\fR
.ad
.sp .6
.RS 4n
\fIclass\fR can be \fBwep\fR or \fBwpa\fR. See preceding discussion.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the creation is temporary. Temporary creation last until the
next reboot.
.RE

.sp
```

```
3424 .ne 2
3425 .na
3426 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
3427 .ad
3428 .sp .6
3429 .RS 4n
3430 See "Options," above.
3431 .RE

3433 .sp
3434 .ne 2
3435 .na
3436 \fB\fB-f\fR \fIfile\fR, \fB--file\fR=\fIfile\fR\fR
3437 .ad
3438 .sp .6
3439 .RS 4n
3440 Specifies a file that should be used to obtain the secure object's value. The
3441 format of this file depends on the secure object class. See the \fBEXAMPLES\fR
3442 section for an example of using this option to set a \fBWEP\fR key.
3443 .RE

3445 .RE

3447 .sp
3448 .ne 2
3449 .na
3450 \fB\fBdladm delete-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
3451 \fIsecobj\fR[,...]\fR
3452 .ad
3453 .sp .6
3454 .RS 4n
3455 Delete one or more specified secure objects. This subcommand is only usable by
3456 users or roles that belong to the "Network Link Security" \fBRBAC\fR profile.
3457 .sp
3458 .ne 2
3459 .na
3460 \fB\fB-t\fR, \fB--temporary\fR\fR
3461 .ad
3462 .sp .6
3463 .RS 4n
3464 Specifies that the deletions are temporary. Temporary deletions last until the
3465 next reboot.
3466 .RE

3468 .sp
3469 .ne 2
3470 .na
3471 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
3472 .ad
3473 .sp .6
3474 .RS 4n
3475 See "Options," above.
3476 .RE

3478 .RE

3480 .sp
3481 .ne 2
3482 .na
3483 \fB\fBdladm show-secobj\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
3484 [\fIsecobj\fR,...]\fR
3485 .ad
3486 .sp .6
3487 .RS 4n
3488 Show current or persistent secure object information. If one or more secure
3489 objects are specified, then information for each is displayed. Otherwise, all
```



```
3490 current or persistent secure objects are displayed.
3491 .sp
3492 By default, current secure objects are displayed, which are all secure objects
3493 that have either been persistently created and not temporarily deleted, or
3494 temporarily created.
3495 .sp
3496 For security reasons, it is not possible to show the value of a secure object.
3497 .sp
3498 .ne 2
3499 .na
3500 \fB\fB-o\fR \fIfield\fR[,...] , \fB--output\fR=\fIfield\fR[,...]\fR
3501 .ad
3502 .sp .6
3503 .RS 4n
3504 A case-insensitive, comma-separated list of output fields to display. The field
3505 name must be one of the fields listed below. For displayed secure object, the
3506 following fields can be shown:
3507 .sp
3508 .ne 2
3509 .na
3510 \fB\fBOBJECT\fR\fR
3511 .ad
3512 .sp .6
3513 .RS 4n
3514 The name of the secure object.
3515 .RE

3517 .sp
3518 .ne 2
3519 .na
3520 \fB\fBCLASS\fR\fR
3521 .ad
3522 .sp .6
3523 .RS 4n
3524 The class of the secure object.
3525 .RE

3527 .RE

3529 .sp
3530 .ne 2
3531 .na
3532 \fB\fB-p\fR, \fB--parseable\fR\fR
3533 .ad
3534 .sp .6
3535 .RS 4n
3536 Display using a stable machine-parseable format. The \fB-o\fR option is
3537 required with \fB-p\fR. See "Parseable Output Format", below.
3538 .RE

3540 .sp
3541 .ne 2
3542 .na
3543 \fB\fB-P\fR, \fB--persistent\fR\fR
3544 .ad
3545 .sp .6
3546 .RS 4n
3547 Display persistent secure object information
3548 .RE

3550 .RE

3552 .sp
3553 .ne 2
3554 .na
3555 \fB\fBdladm create-vnic\fR [\fB-t\fR] \fB-l\fR \fIlink\fR [\fB-R\fR
```

```
\fIroot-dir\fR] [\fB-m\fR \fIvalue\fR | auto | {factory [\fB-n\fR
\fIslot-identifier\fR]} | {random [\fB-r\fR \fIprefix\fR]}] [\fB-v\fR
\fIvlan-id\fR] [\fB-p\fR \fIprop\fR=\fIvalue\fR[,...]] \fIvnic-link\fR\fR
.ad
.sp .6
.RS 4n
Create a VNIC with name \fIvnic-link\fR over the specified link.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the VNIC is temporary. Temporary VNICs last until the next
reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-l\fR \fIlink\fR, \fB--link\fR=\fIlink\fR\fR
.ad
.sp .6
.RS 4n
\fIlink\fR can be a physical link or an \fBetherstub\fR.
.RE

.sp
.ne 2
.na
\fB\fB-m\fR \fIvalue\fR | \fIkeyword\fR, \fB--mac-address\fR=\fIvalue\fR |
\fIkeyword\fR\fR
.ad
.sp .6
.RS 4n
Sets the VNIC's MAC address based on the specified value or keyword. If
\fIvalue\fR is not a keyword, it is interpreted as a unicast MAC address, which
must be valid for the underlying NIC. The following special keywords can be
used:
.sp
.ne 2
.na
\fBfactory [\fB-n\fR \fIslot-identifier\fR],\fR
.ad
.br
.na
\fBfactory [\fB--slot\fR=\fIslot-identifier\fR]\fR
.ad
.sp .6
.RS 4n
Assign a factory MAC address to the VNIC. When a factory MAC address is
requested, \fB-m\fR can be combined with the \fB-n\fR option to specify a MAC
address slot to be used. If \fB-n\fR is not specified, the system will choose
the next available factory MAC address. The \fB-m\fR option of the
\fBshow-phys\fR subcommand can be used to display the list of factory MAC
```



```
addresses, their slot identifiers, and their availability.
.RE

.sp
.ne 2
.na
\fB\fR
.ad
.br
.na
\fBrandom [\fB-r\fR \fIprefix\fR],\fR
.ad
.br
.na
\fBrandom [\fB--mac-prefix\fR=\fIprefix\fR]\fR
.ad
.sp .6
.RS 4n
Assign a random MAC address to the VNIC. A default prefix consisting of a valid
IEEE OUI with the local bit set will be used. That prefix can be overridden
with the \fB-r\fR option.
.RE

.sp
.ne 2
.na
\fBauto\fR
.ad
.sp .6
.RS 4n
Try and use a factory MAC address first. If none is available, assign a random
MAC address. \fBauto\fR is the default action if the \fB-m\fR option is not
specified.
.RE

.sp
.ne 2
.na
\fB\fB-v\fR \fIvlan-id\fR\fR
.ad
.sp .6
.RS 4n
Enable VLAN tagging for this VNIC. The VLAN tag will have id \fIvlan-id\fR.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIprop\fR=\fIvalue\fR,..., \fB--prop\fR
\fIprop\fR=\fIvalue\fR,...\fR
.ad
.sp .6
.RS 4n
A comma-separated list of properties to set to the specified values.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm delete-vnic\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIvnic-link\fR\fR
.ad
```

```
3688 .sp .6
3689 .RS 4n
3690 Deletes the specified VNIC.
3691 .sp
3692 .ne 2
3693 .na
3694 \fB\fB-t\fR, \fB--temporary\fR\fR
3695 .ad
3696 .sp .6
3697 .RS 4n
3698 Specifies that the deletion is temporary. Temporary deletions last until the
3699 next reboot.
3700 .RE

3702 .sp
3703 .ne 2
3704 .na
3705 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
3706 .ad
3707 .sp .6
3708 .RS 4n
3709 See "Options," above.
3710 .RE

3712 .RE

3714 .sp
3715 .ne 2
3716 .na
3717 \fB\fBdladm show-vnic\fR [\fB-pP\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
3718 [\fB-o\fR \fIfield\fR[,...]] [\fB-l\fR \fIlink\fR] [\fIvnic-link\fR]\fR
3719 .ad
3720 .sp .6
3721 .RS 4n
3722 Show VNIC configuration information (the default) or statistics, for all VNICs,
3723 all VNICs on a link, or only the specified \fIvnic-link\fR.
3724 .sp
3725 .ne 2
3726 .na
3727 \fB\fB-o\fR \fIfield\fR[,...] , \fB--output\fR=\fIfield\fR[,...]\fR
3728 .ad
3729 .sp .6
3730 .RS 4n
3731 A case-insensitive, comma-separated list of output fields to display. The field
3732 name must be one of the fields listed below. The field name must be one of the
3733 fields listed below, or the special value \fBall\fR to display all fields. By
3734 default (without \fB-o\fR), \fBshow-vnic\fR displays all fields.
3735 .sp
3736 .ne 2
3737 .na
3738 \fB\fBLINK\fR\fR
3739 .ad
3740 .sp .6
3741 .RS 4n
3742 The name of the VNIC.
3743 .RE

3745 .sp
3746 .ne 2
3747 .na
3748 \fB\fBOVER\fR\fR
3749 .ad
3750 .sp .6
3751 .RS 4n
3752 The name of the physical link over which this VNIC is configured.
3753 .RE
```



```
3755 .sp
3756 .ne 2
3757 .na
3758 \fB\fBSPEED\fR\fR
3759 .ad
3760 .sp .6
3761 .RS 4n
3762 The maximum speed of the VNIC, in megabits per second.
3763 .RE

3765 .sp
3766 .ne 2
3767 .na
3768 \fB\fBMACADDRESS\fR\fR
3769 .ad
3770 .sp .6
3771 .RS 4n
3772 MAC address of the VNIC.
3773 .RE

3775 .sp
3776 .ne 2
3777 .na
3778 \fB\fBMACADDRTYPE\fR\fR
3779 .ad
3780 .sp .6
3781 .RS 4n
3782 MAC address type of the VNIC. \fBdladm\fR distinguishes among the following MAC
3783 address types:
3784 .sp
3785 .ne 2
3786 .na
3787 \fB\fBrandom\fR\fR
3788 .ad
3789 .sp .6
3790 .RS 4n
3791 A random address assigned to the VNIC.
3792 .RE

3794 .sp
3795 .ne 2
3796 .na
3797 \fB\fBfactory\fR\fR
3798 .ad
3799 .sp .6
3800 .RS 4n
3801 A factory MAC address used by the VNIC.
3802 .RE

3804 .RE

3806 .RE

3808 .sp
3809 .ne 2
3810 .na
3811 \fB\fB-p\fR, \fB--parseable\fR\fR
3812 .ad
3813 .sp .6
3814 .RS 4n
3815 Display using a stable machine-parseable format. The \fB-o\fR option is
3816 required with \fB-p\fR. See "Parseable Output Format", below.
3817 .RE

3819 .sp
```

```
.ne 2
.na
\fB\fB-P\fR, \fB--persistent\fR\fR
.ad
.sp .6
.RS 4n
Display the persistent VNIC configuration.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR, \fB--statistics\fR\fR
.ad
.sp .6
.RS 4n
Displays VNIC statistics.
.RE

.sp
.ne 2
.na
\fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
.ad
.sp .6
.RS 4n
Used with the \fB-s\fR option to specify an interval, in seconds, at which
statistics should be displayed. If this option is not specified, statistics
will be displayed only once.
.RE

.sp
.ne 2
.na
\fB\fB-l\fR \fIlink\fR, \fB--link\fR=\fIlink\fR\fR
.ad
.sp .6
.RS 4n
Display information for all VNICs on the named link.
.RE

.RE

.sp
.ne 2
.na
\fB\fR
.ad
.br
.na
\fB\fBdladm create-etherstub\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIetherstub\fR\fR
.ad
.sp .6
.RS 4n
Create an etherstub with the specified name.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the etherstub is temporary. Temporary etherstubs do not persist
across reboots.
.RE
```



```
.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

VNICs can be created on top of etherstubs instead of physical NICs. As with
physical NICs, such a creation causes the stack to implicitly create a virtual
switch between the VNICs created on top of the same etherstub.
.RE

.sp
.ne 2
.na
\fB\fR
.ad
.br
.na
\fB\fBdladm delete-etherstub\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIetherstub\fR\fR
.ad
.sp .6
.RS 4n
Delete the specified etherstub.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the deletion is temporary. Temporary deletions last until the
next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm show-etherstub\fR [\fIetherstub\fR]\fR
.ad
.sp .6
.RS 4n
Show all configured etherstubs by default, or the specified etherstub if
\fIetherstub\fR is specified.
.RE

.sp
.ne 2
.na
```

\fB\fBdladm create-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-T\fR
\fItype\fR [\fB-s\fR \fItsrc\fR] [\fB-d\fR \fItdst\fR] \fIiptun-link\fR\fR
.ad
.sp .6
.RS 4n
Create an IP tunnel link named \fIiptun-link\fR. Such links can additionally be protected with IPsec using \fBipsecconf\fR(1M).
.sp
An IP tunnel is conceptually comprised of two parts: a virtual link between two or more IP nodes, and an IP interface above this link that allows the system to transmit and receive IP packets encapsulated by the underlying link. This subcommand creates a virtual link. The \fBifconfig\fR(1M) command is used to configure IP interfaces above the link.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the IP tunnel link is temporary. Temporary tunnels last until the next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-T\fR \fItype\fR, \fB--tunnel-type\fR=\fItype\fR\fR
.ad
.sp .6
.RS 4n
Specifies the type of tunnel to be created. The type must be one of the following:
.sp
.ne 2
.na
\fB\fBipv4\fR\fR
.ad
.sp .6
.RS 4n
A point-to-point, IP-over-IP tunnel between two IPv4 nodes. This type of tunnel requires IPv4 source and destination addresses to function. IPv4 and IPv6 interfaces can be plumbed above such a tunnel to create IPv4-over-IPv4 and IPv6-over-IPv4 tunneling configurations.
.RE

.sp
.ne 2
.na
\fB\fBipv6\fR\fR
.ad
.sp .6
.RS 4n
A point-to-point, IP-over-IP tunnel between two IPv6 nodes as defined in IETF RFC 2473. This type of tunnel requires IPv6 source and destination addresses to function. IPv4 and IPv6 interfaces can be plumbed above such a tunnel to create

IPv4-over-IPv6 and IPv6-over-IPv6 tunneling configurations.
.RE

.sp
.ne 2
.na
\fB\fB6to4\fR\fR
.ad
.sp .6
.RS 4n
A 6to4, point-to-multipoint tunnel as defined in IETF RFC 3056. This type of tunnel requires an IPv4 source address to function. An IPv6 interface is plumbed on such a tunnel link to configure a 6to4 router.
.RE

.RE

.sp
.ne 2
.na
\fB\fB-s\fR \fItsrc\fR, \fB--tunnel-src\fR=\fItsrc\fR\fR
.ad
.sp .6
.RS 4n
Literal IP address or hostname corresponding to the tunnel source. If a hostname is specified, it will be resolved to IP addresses, and one of those IP addresses will be used as the tunnel source. Because IP tunnels are created before naming services have been brought online during the boot process, it is important that any hostname used be included in \fB/etc/hosts\fR.
.RE

.sp
.ne 2
.na
\fB\fB-d\fR \fItdst\fR, \fB--tunnel-dst\fR=\fItdst\fR\fR
.ad
.sp .6
.RS 4n
Literal IP address or hostname corresponding to the tunnel destination.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm modify-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-s\fR \fItsrc\fR] [\fB-d\fR \fItdst\fR] \fIiptun-link\fR\fR
.ad
.sp .6
.RS 4n
Modify the parameters of the specified IP tunnel.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the modification is temporary. Temporary modifications last until the next reboot.
.RE

.sp
.ne 2
.na

```
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR \fItsrc\fR, \fB--tunnel-src\fR=\fItsrc\fR\fR
.ad
.sp .6
.RS 4n
Specifies a new tunnel source address. See \fBcreate-iptun\fR for a
description.
.RE

.sp
.ne 2
.na
\fB\fB-d\fR \fItdst\fR, \fB--tunnel-dst\fR=\fItdst\fR\fR
.ad
.sp .6
.RS 4n
Specifies a new tunnel destination address. See \fBcreate-iptun\fR for a
description.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm delete-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIiptun-link\fR\fR
.ad
.sp .6
.RS 4n
Delete the specified IP tunnel link.
.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the deletion is temporary. Temporary deletions last until the
next reboot.
.RE

.sp
.ne 2
.na
\fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
.ad
.sp .6
.RS 4n
See "Options," above.
.RE

.RE

.sp
.ne 2
.na
```



```
\fB\fBdladm show-iptun\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
[\fIiptun-link\fR]\fR
.ad
.sp .6
.RS 4n
Show IP tunnel link configuration for a single IP tunnel or all IP tunnels.
.sp
.ne 2
.na
\fB\fB-P\fR, \fB--persistent\fR\fR
.ad
.sp .6
.RS 4n
Display the persistent IP tunnel configuration.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Display using a stable machine-parseable format. The -o option is required with
-p. See "Parseable Output Format", below.
.RE

.sp
.ne 2
.na
\fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
.ad
.sp .6
.RS 4n
A case-insensitive, comma-separated list of output fields to display. The field
name must be one of the fields listed below, or the special value \fBall\fR, to
display all fields. By default (without \fB-o\fR), \fBshow-iptun\fR displays
all fields.
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the IP tunnel link.
.RE

.sp
.ne 2
.na
\fB\fBTYPE\fR\fR
.ad
.sp .6
.RS 4n
Type of tunnel as specified by the \fB-T\fR option of \fBcreate-iptun\fR.
.RE

.sp
.ne 2
.na
\fB\fBFLAGS\fR\fR
.ad
.sp .6
.RS 4n
A set of flags associated with the IP tunnel link. Possible flags are:
```

```
.sp
.ne 2
.na
\fB\fBs\fR\fR
.ad
.sp .6
.RS 4n
The IP tunnel link is protected by IPsec policy. To display the IPsec policy
associated with the tunnel link, enter:
.sp
.in +2
.nf
# \fBipsecconf -ln -i \fItunnel-link\fR\fR
.fi
.in -2
.sp

See \fBipsecconf\fR(1M) for more details on how to configure IPsec policy.
.RE

.sp
.ne 2
.na
\fB\fBi\fR\fR
.ad
.sp .6
.RS 4n
The IP tunnel link was implicitly created with \fBifconfig\fR(1M), and will be
automatically deleted when it is no longer referenced (that is, when the last
IP interface over the tunnel is unplumbed). See \fBifconfig\fR(1M) for details
on implicit tunnel creation.
.RE

.RE

.sp
.ne 2
.na
\fB\fBSOURCE\fR\fR
.ad
.sp .6
.RS 4n
The tunnel source address.
.RE

.sp
.ne 2
.na
\fB\fBDESTINATION\fR\fR
.ad
.sp .6
.RS 4n
The tunnel destination address.
.RE

.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm show-usage\fR [\fB-a\fR] \fB-f\fR \fIfilename\fR [\fB-p\fR
\fIplotfile\fR \fB-F\fR \fIformat\fR] [\fB-s\fR \fItime\fR] [\fB-e\fR
\fItime\fR] [\fIlink\fR]\fR
.ad
```



```
.sp .6
.RS 4n
Show the historical network usage from a stored extended accounting file.
Configuration and enabling of network accounting through \fBacctadm\fR(1M) is
required. The default output will be the summary of network usage for the
entire period of time in which extended accounting was enabled.
.sp
.ne 2
.na
\fB\fB-a\fR\fR
.ad
.sp .6
.RS 4n
Display all historical network usage for the specified period of time during
which extended accounting is enabled. This includes the usage information for
the links that have already been deleted.
.RE

.sp
.ne 2
.na
\fB\fB-f\fR \fIfilename\fR, \fB--file\fR=\fIfilename\fR\fR
.ad
.sp .6
.RS 4n
Read extended accounting records of network usage from \fIfilename\fR.
.RE

.sp
.ne 2
.na
\fB\fB-F\fR \fIformat\fR, \fB--format\fR=\fIformat\fR\fR
.ad
.sp .6
.RS 4n
Specifies the format of \fIplotfile\fR that is specified by the \fB-p\fR
option. As of this release, \fBgnuplot\fR is the only supported format.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR \fIplotfile\fR, \fB--plot\fR=\fIplotfile\fR\fR
.ad
.sp .6
.RS 4n
Write network usage data to a file of the format specified by the \fB-F\fR
option, which is required.
.RE

.sp
.ne 2
.na
\fB\fB-s\fR \fItime\fR, \fB--start\fR=\fItime\fR\fR
.ad
.br
.na
\fB\fB-e\fR \fItime\fR, \fB--stop\fR=\fItime\fR\fR
.ad
.sp .6
.RS 4n
Start and stop times for data display. Time is in the format
\fIMM\fR/\fIDD\fR/\fIYYYY\fR,\fIhh\fR:\fImm\fR:\fIss\fR.
.RE

.sp
```

```
.ne 2
.na
\fB\fIlink\fR\fR
.ad
.sp .6
.RS 4n
If specified, display the network usage only for the named link. Otherwise,
display network usage for all links.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm help\fR [\fIsubcommand\fR]\fR
.ad
.sp .6
.RS 4n
Displays all subcommands or help on a single subcommand.
.RE

#endif /* ! codereview */
.SS "Parseable Output Format"
.sp
.LP
Many \fBdladm\fR subcommands have an option that displays output in a
machine-parseable format. The output format is one or more lines of colon
(\fB:\fR) delimited fields. The fields displayed are specific to the subcommand
used and are listed under the entry for the \fB-o\fR option for a given
subcommand. Output includes only those fields requested by means of the
\fB-o\fR option, in the order requested.
.sp
.LP
When you request multiple fields, any literal colon characters are escaped by a
backslash (\fB\e\fR) before being output. Similarly, literal backslash
characters will also be escaped (\fB\e\e\fR). This escape format is parseable
by using shell \fBread\fR(1) functions with the environment variable
\fBIFS=:\fR (see \fBEXAMPLES\fR, below). Note that escaping is not done when
you request only a single field.
.SS "General Link Properties"
.sp
.LP
The following general link properties are supported:
.sp
.ne 2
.na
\fB\fBautopush\fR\fR
.ad
.sp .6
.RS 4n
Specifies the set of STREAMS modules to push on the stream associated with a
link when its DLPI device is opened. It is a space-delimited list of modules.
.sp
The optional special character sequence \fB[anchor]\fR indicates that a STREAMS
anchor should be placed on the stream at the module previously specified in the
list. It is an error to specify more than one anchor or to have an anchor first
in the list.
.sp
The \fBautopush\fR property is preferred over the more general
\fBautopush\fR(1M) command.
.RE

.sp
.ne 2
.na
```



```
\fB\fBcpus\fR\fR
.ad
.sp .6
.RS 4n
Bind the processing of packets for a given data link to a processor or a set of
processors. The value can be a comma-separated list of one or more processor
ids. If the list consists of more than one processor, the processing will
spread out to all the processors. Connection to processor affinity and packet
ordering for any individual connection will be maintained.
.sp
The processor or set of processors are not exclusively reserved for the link.
Only the kernel threads and interrupts associated with processing of the link
are bound to the processor or the set of processors specified. In case it is
desired that processors be dedicated to the link, \fBpsrset\fR(1M) can be used
to create a processor set and then specifying the processors from the processor
set to bind the link to.
.sp
If the link was already bound to processor or set of processors due to a
previous operation, the binding will be removed and the new set of processors
will be used instead.
.sp
The default is no CPU binding, which is to say that the processing of packets
is not bound to any specific processor or processor set.
.RE

.sp
.ne 2
.na
\fB\fBlearn_limit\fR\fR
.ad
.sp .6
.RS 4n
Limits the number of new or changed MAC sources to be learned over a bridge
link. When the number exceeds this value, learning on that link is temporarily
disabled. Only non-VLAN, non-VNIC type links have this property.
.sp
The default value is \fB1000\fR. Valid values are greater or equal to 0.
.RE

.sp
.ne 2
.na
\fB\fBlearn_decay\fR\fR
.ad
.sp .6
.RS 4n
Specifies the decay rate for source changes limited by \fBlearn_limit\fR. This
number is subtracted from the counter for a bridge link every 5 seconds. Only
non-VLAN, non-VNIC type links have this property.
.sp
The default value is \fB200\fR. Valid values are greater or equal to 0.
.RE

.sp
.ne 2
.na
\fB\fBmaxbw\fR\fR
.ad
.sp .6
.RS 4n
Sets the full duplex bandwidth for the link. The bandwidth is specified as an
integer with one of the scale suffixes (\fBK\fR, \fBM\fR, or \fBG\fR for Kbps,
Mbps, and Gbps). If no units are specified, the input value will be read as
Mbps. The default is no bandwidth limit.
.RE
```

```
.sp
.ne 2
.na
\fB\fBpriority\fR\fR
.ad
.sp .6
.RS 4n
Sets the relative priority for the link. The value can be given as one of the
tokens \fBhigh\fR, \fBmedium\fR, or \fBlow\fR. The default is \fBhigh\fR.
.RE

.sp
.ne 2
.na
\fB\fBstp\fR\fR
.ad
.sp .6
.RS 4n
Enables or disables Spanning Tree Protocol on a bridge link. Setting this value
to \fB0\fR disables Spanning Tree, and puts the link into forwarding mode with
BPDU guarding enabled. This mode is appropriate for point-to-point links
connected only to end nodes. Only non-VLAN, non-VNIC type links have this
property. The default value is \fB1\fR, to enable STP.
.RE

.sp
.ne 2
.na
\fB\fBforward\fR\fR
.ad
.sp .6
.RS 4n
Enables or disables forwarding for a VLAN. Setting this value to \fB0\fR
disables bridge forwarding for a VLAN link. Disabling bridge forwarding removes
that VLAN from the "allowed set" for the bridge. The default value is \fB1\fR,
to enable bridge forwarding for configured VLANs.
.RE

.sp
.ne 2
.na
\fB\fBdefault_tag\fR\fR
.ad
.sp .6
.RS 4n
Sets the default VLAN ID that is assumed for untagged packets sent to and
received from this link. Only non-VLAN, non-VNIC type links have this property.
Setting this value to \fB0\fR disables the bridge forwarding of untagged
packets to and from the port. The default value is \fBVLAN ID 1\fR. Valid
values values are from 0 to 4094.
.RE

.sp
.ne 2
.na
\fB\fBstp_priority\fR\fR
.ad
.sp .6
.RS 4n
Sets the STP and RSTP Port Priority value, which is used to determine the
preferred root port on a bridge. Lower numerical values are higher priority.
The default value is \fB128\fR. Valid values range from 0 to 255.
.RE

.sp
.ne 2
```



```
.na
\fB\fBstp_cost\fR\fR
.ad
.sp .6
.RS 4n
Sets the STP and RSTP cost for using the link. The default value is \fBauto\fR,
which sets the cost based on link speed, using \fB100\fR for 10Mbps, \fB19\fR
for 100Mbps, \fB4\fR for 1Gbps, and \fB2\fR for 10Gbps. Valid values range from
1 to 65535.
.RE

.sp
.ne 2
.na
\fB\fBstp_edge\fR\fR
.ad
.sp .6
.RS 4n
Enables or disables bridge edge port detection. If set to \fB0\fR (false), the
system assumes that the port is connected to other bridges even if no bridge
PDUs of any type are seen. The default value is \fB1\fR, which detects edge
ports automatically.
.RE

.sp
.ne 2
.na
\fB\fBstp_p2p\fR\fR
.ad
.sp .6
.RS 4n
Sets bridge point-to-point operation mode. Possible values are \fBtrue\fR,
\fBfalse\fR, and \fBauto\fR. When set to \fBauto\fR, point-to-point connections
are automatically discovered. When set to \fBtrue\fR, the port mode is forced
to use point-to-point. When set to \fBfalse\fR, the port mode is forced to use
normal multipoint mode. The default value is \fBauto\fR.
.RE

.sp
.ne 2
.na
\fB\fBstp_mcheck\fR\fR
.ad
.sp .6
.RS 4n
Triggers the system to run the RSTP \fBForce BPDU Migration Check\fR procedure
on this link. The procedure is triggered by setting the property value to
\fB1\fR. The property is automatically reset back to \fB0\fR. This value cannot
be set unless the following are true:
.RS +4
.TP
.ie t \(bu
.el o
The link is bridged
.RE
.RS +4
.TP
.ie t \(bu
.el o
The bridge is protected by Spanning Tree
.RE
.RS +4
.TP
.ie t \(bu
.el o
The bridge \fBforce-protocol\fR value is at least 2 (RSTP)
```

```
.RE
The default value is 0.
.RE

.sp
.ne 2
.na
\fB\fBzone\fR\fR
.ad
.sp .6
.RS 4n
Specifies the zone to which the link belongs. This property can be modified
only temporarily through \fBdladm\fR, and thus the \fB-t\fR option must be
specified. To modify the zone assignment such that it persists across reboots,
please use \fBzonecfg\fR(1M). Possible values consist of any exclusive-IP zone
currently running on the system. By default, the zone binding is as per
\fBzonecfg\fR(1M).
.RE

.SS "Wifi Link Properties"
.sp
.LP
The following \fBWiFi\fR link properties are supported. Note that the ability
to set a given property to a given value depends on the driver and hardware.
.sp
.ne 2
.na
\fB\fBchannel\fR\fR
.ad
.sp .6
.RS 4n
Specifies the channel to use. This property can be modified only by certain
\fBWiFi\fR links when in \fBIBSS\fR mode. The default value and allowed range
of values varies by regulatory domain.
.RE

.sp
.ne 2
.na
\fB\fBpowermode\fR\fR
.ad
.sp .6
.RS 4n
Specifies the power management mode of the \fBWiFi\fR link. Possible values are
\fBoff\fR (disable power management), \fBmax\fR (maximum power savings), and
\fBfast\fR (performance-sensitive power management). Default is \fBoff\fR.
.RE

.sp
.ne 2
.na
\fB\fBradio\fR\fR
.ad
.sp .6
.RS 4n
Specifies the radio mode of the \fBWiFi\fR link. Possible values are \fBon\fR
or \fBoff\fR. Default is \fBon\fR.
.RE

.sp
.ne 2
.na
\fB\fBspeed\fR\fR
.ad
.sp .6
.RS 4n
```



```
Specifies a fixed speed for the \fBWiFi\fR link, in megabits per second. The
set of possible values depends on the driver and hardware (but is shown by
\fBshow-linkprop\fR); common speeds include 1, 2, 11, and 54. By default, there
is no fixed speed.
.RE

.SS "Ethernet Link Properties"
.sp
.LP
The following MII Properties, as documented in \fBieee802.3\fR(5), are
supported in read-only mode:
.RS +4
.TP
.ie t \(bu
.el o
\fBduplex\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBstate\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_autoneg_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_10gfdx_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_1000fdx_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_1000hdx_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_100fdx_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_100hdx_cap\fR
.RE
.RS +4
.TP
.ie t \(bu
.el o
\fBadv_10fdx_cap\fR
.RE
.RS +4
```

```
.TP
.ie t \(bu
.el o
\fBadv_10hdx_cap\fR
.RE
.sp
.LP
Each \fBadv_\fR property (for example, \fBadv_10fdx_cap\fR) also has a
read/write counterpart \fBen_\fR property (for example, \fBen_10fdx_cap\fR)
controlling parameters used at auto-negotiation. In the absence of Power
Management, the \fBadv\fR* speed/duplex parameters provide the values that are
both negotiated and currently effective in hardware. However, with Power
Management enabled, the speed/duplex capabilities currently exposed in hardware
might be a subset of the set of bits that were used in initial link parameter
negotiation. Thus the MII \fBadv_\fR* parameters are marked read-only, with an
additional set of \fBen_\fR* parameters for configuring speed and duplex
properties at initial negotiation.
.sp
.LP
Note that the \fBadv_autoneg_cap\fR does not have an \fBen_autoneg_cap\fR
counterpart: the \fBadv_autoneg_cap\fR is a 0/1 switch that turns off/on
autonegotiation itself, and therefore cannot be impacted by Power Management.
.sp
.LP
In addition, the following Ethernet properties are reported:
.sp
.ne 2
.na
\fB\fBspeed\fR\fR
.ad
.sp .6
.RS 4n
(read-only) The operating speed of the device, in Mbps.
.RE

.sp
.ne 2
.na
\fB\fBmtu\fR\fR
.ad
.sp .6
.RS 4n
The maximum client SDU (Send Data Unit) supported by the device. Valid range is
68-65536.
.RE

.sp
.ne 2
.na
\fB\fBflowctrl\fR\fR
.ad
.sp .6
.RS 4n
Establishes flow-control modes that will be advertised by the device. Valid
input is one of:
.sp
.ne 2
.na
\fB\fBno\fR\fR
.ad
.sp .6
.RS 4n
No flow control enabled.
.RE

.sp
```



```
.ne 2
.na
\fB\fBrx\fR\fR
.ad
.sp .6
.RS 4n
Receive, and act upon incoming pause frames.
.RE

.sp
.ne 2
.na
\fB\fBtx\fR\fR
.ad
.sp .6
.RS 4n
Transmit pause frames to the peer when congestion occurs, but ignore received
pause frames.
.RE

.sp
.ne 2
.na
\fB\fBbi\fR\fR
.ad
.sp .6
.RS 4n
Bidirectional flow control.
.RE

Note that the actual settings for this value are constrained by the
capabilities allowed by the device and the link partner.
.RE

.sp
.ne 2
.na
\fB\fBtagmode\fR\fR
.ad
.sp .6
.RS 4n
This link property controls the conditions in which 802.1Q VLAN tags will be
inserted in packets being transmitted on the link. Two mode values can be
assigned to this property:
.sp
.ne 2
.na
\fB\fBnormal\fR\fR
.ad
.RS 12n
Insert a VLAN tag in outgoing packets under the following conditions:
.RS +4
.TP
.ie t \(bu
.el o
The packet belongs to a VLAN.
.RE
.RS +4
.TP
.ie t \(bu
.el o
The user requested priority tagging.
.RE
.RE

.sp
```

```
.ne 2
.na
\fB\fBvlanonly\fR\fR
.ad
.RS 12n
Insert a VLAN tag only when the outgoing packet belongs to a VLAN. If a tag is
being inserted in this mode and the user has also requested a non-zero
priority, the priority is honored and included in the VLAN tag.
.RE

The default value is \fBvlanonly\fR.
.RE

.SS "IP Tunnel Link Properties"
.sp
.LP
The following IP tunnel link properties are supported.
.sp
.ne 2
.na
\fB\fBhoplimit\fR\fR
.ad
.sp .6
.RS 4n
Specifies the IPv4 TTL or IPv6 hop limit for the encapsulating outer IP header
of a tunnel link. This property exists for all tunnel types. The default value
is 64.
.RE

.sp
.ne 2
.na
\fB\fBencaplimit\fR\fR
.ad
.sp .6
.RS 4n
Specifies the IPv6 encapsulation limit for an IPv6 tunnel as defined in RFC
2473. This value is the tunnel nesting limit for a given tunneled packet. The
default value is 4. A value of 0 disables the encapsulation limit.
.RE

.SH EXAMPLES
.LP
\fBExample 1 \fRConfiguring an Aggregation
.sp
.LP
To configure a data-link over an aggregation of devices \fBbge0\fR and
\fBbge1\fR with key 1, enter the following command:

.sp
.in +2
.nf
# \fBdladm create-aggr -d bge0 -d bge1 1\fR
.fi
.in -2
.sp

.LP
\fBExample 2 \fRConnecting to a WiFi Link
.sp
.LP
To connect to the most optimal available unsecured network on a system with a
single \fBWiFi\fR link (as per the prioritization rules specified for
\fBconnect-wifi\fR), enter the following command:

.sp
```



```
.in +2
.nf
# \fBdladm connect-wifi\fR
.fi
.in -2
.sp

.LP
\fBExample 3 \fRCreating a WiFi Key
.sp
.LP
To interactively create the \fBWEP\fR key \fBmykey\fR, enter the following
command:

.sp
.in +2
.nf
# \fBdladm create-secobj -c wep mykey\fR
.fi
.in -2
.sp

.sp
.LP
Alternatively, to non-interactively create the \fBWEP\fR key \fBmykey\fR using
the contents of a file:

.sp
.in +2
.nf
# \fBumask 077\fR
 # \fBcat >/tmp/mykey.$$ <<EOF\fR
 \fB12345\fR
 \fBEOF\fR
 # \fBdladm create-secobj -c wep -f /tmp/mykey.$$ mykey\fR
 # \fBrm /tmp/mykey.$$\fR
.fi
.in -2
.sp

.LP
\fBExample 4 \fRConnecting to a Specified Encrypted WiFi Link
.sp
.LP
To use key \fBmykey\fR to connect to \fBESSID\fR \fBwlan\fR on link \fBath0\fR,
enter the following command:

.sp
.in +2
.nf
# \fBdladm connect-wifi -k mykey -e wlan ath0\fR
.fi
.in -2
.sp

.LP
\fBExample 5 \fRChanging a Link Property
.sp
.LP
To set \fBpowermode\fR to the value \fBfast\fR on link \fBpcwl0\fR, enter the
following command:

.sp
.in +2
.nf
# \fBdladm set-linkprop -p powermode=fast pcwl0\fR
```

```
.fi
.in -2
.sp

.LP
\fBExample 6 \fRConnecting to a WPA-Protected WiFi Link
.sp
.LP
Create a WPA key \fBpsk\fR and enter the following command:

.sp
.in +2
.nf
# \fBdladm create-secobj -c wpa psk\fR
.fi
.in -2
.sp

.sp
.LP
To then use key \fBpsk\fR to connect to ESSID \fBwlan\fR on link \fBath0\fR,
enter the following command:

.sp
.in +2
.nf
# \fBdladm connect-wifi -k psk -e wlan ath0\fR
.fi
.in -2
.sp

.LP
\fBExample 7 \fRRenaming a Link
.sp
.LP
To rename the \fBbge0\fR link to \fBmgmt0\fR, enter the following command:

.sp
.in +2
.nf
# \fBdladm rename-link bge0 mgmt0\fR
.fi
.in -2
.sp

.LP
\fBExample 8 \fRReplacing a Network Card
.sp
.LP
Consider that the \fBbge0\fR device, whose link was named \fBmgmt0\fR as shown
in the previous example, needs to be replaced with a \fBce0\fR device because
of a hardware failure. The \fBbge0\fR NIC is physically removed, and replaced
with a new \fBce0\fR NIC. To associate the newly added \fBce0\fR device with
the \fBmgmt0\fR configuration previously associated with \fBbge0\fR, enter the
following command:

.sp
.in +2
.nf
# \fBdladm rename-link ce0 mgmt0\fR
.fi
.in -2
.sp

.LP
\fBExample 9 \fRRemoving a Network Card
```



```
.sp
.LP
Suppose that in the previous example, the intent is not to replace the
\fBbge0\fR NIC with another NIC, but rather to remove and not replace the
hardware. In that case, the \fBmgmt0\fR datalink configuration is not slated to
be associated with a different physical device as shown in the previous
example, but needs to be deleted. Enter the following command to delete the
datalink configuration associated with the \fBmgmt0\fR datalink, whose physical
hardware (\fBbge0\fR in this case) has been removed:

.sp
.in +2
.nf
# \fBdladm delete-phys mgmt0\fR
.fi
.in -2
.sp

.LP
\fBExample 10 \fRUsing Parseable Output to Capture a Single Field
.sp
.LP
The following assignment saves the MTU of link \fBnet0\fR to a variable named
\fBmtu\fR.

.sp
.in +2
.nf
# \fBmtu=‘dladm show-link -p -o mtu net0‘\fR
.fi
.in -2
.sp

.LP
\fBExample 11 \fRUsing Parseable Output to Iterate over Links
.sp
.LP
The following script displays the state of each link on the system.

.sp
.in +2
.nf
# \fBdladm show-link -p -o link,state | while IFS=: read link state; do
            print "Link $link is in state $state"
        done\fR
.fi
.in -2
.sp

.LP
\fBExample 12 \fRConfiguring VNICs
.sp
.LP
Create two VNICs with names \fBhello0\fR and \fBtest1\fR over a single physical
link \fBbge0\fR:

.sp
.in +2
.nf
# \fBdladm create-vnic -l bge0 hello0\fR
# \fBdladm create-vnic -l bge0 test1\fR
.fi
.in -2
.sp

.LP
```

```
\fBExample 13 \fRConfiguring VNICs and Allocating Bandwidth and Priority
.sp
.LP
Create two VNICs with names \fBhello0\fR and \fBtest1\fR over a single physical
link \fBbge0\fR and make \fBhello0\fR a high priority VNIC with a
factory-assigned MAC address with a maximum bandwidth of 50 Mbps. Make
\fBtest1\fR a low priority VNIC with a random MAC address and a maximum
bandwidth of 100Mbps.

.sp
.in +2
.nf
# \fBdladm create-vnic -l bge0 -m factory -p maxbw=50,priority=high hello0\fR
# \fBdladm create-vnic -l bge0 -m random -p maxbw=100M,priority=low test1\fR
.fi
.in -2
.sp

.LP
\fBExample 14 \fRConfiguring a VNIC with a Factory MAC Address
.sp
.LP
First, list the available factory MAC addresses and choose one of them:

.sp
.in +2
.nf
# \fBdladm show-phys -m bge0\fR
LINK            SLOT         ADDRESS             INUSE    CLIENT
bge0            primary      0:e0:81:27:d4:47    yes      bge0
bge0            1            8:0:20:fe:4e:a5     no
bge0            2            8:0:20:fe:4e:a6     no
bge0            3            8:0:20:fe:4e:a7     no
.fi
.in -2
.sp

.sp
.LP
Create a VNIC named \fBhello0\fR and use slot 1's address:

.sp
.in +2
.nf
# \fBdladm create-vnic -l bge0 -m factory -n 1 hello0\fR
# \fBdladm show-phys -m bge0\fR
LINK            SLOT         ADDRESS             INUSE    CLIENT
bge0            primary      0:e0:81:27:d4:47    yes      bge0
bge0            1            8:0:20:fe:4e:a5     yes      hello0
bge0            2            8:0:20:fe:4e:a6     no
bge0            3            8:0:20:fe:4e:a7     no
.fi
.in -2
.sp

.LP
\fBExample 15 \fRCreating a VNIC with User-Specified MAC Address, Binding it to
Set of Processors
.sp
.LP
Create a VNIC with name \fBhello0\fR, with a user specified MAC address, and a
processor binding \fB0, 1, 2, 3\fR.

.sp
.in +2
.nf
```



```
# \fBdladm create-vnic -l bge0 -m 8:0:20:fe:4e:b8 -p cpus=0,1,2,3 hello0\fR
.fi
.in -2
.sp

.LP
\fBExample 16 \fRCreating a Virtual Network Without a Physical NIC
.sp
.LP
First, create an etherstub with name \fBstub1\fR:

.sp
.in +2
.nf
# \fBdladm create-etherstub stub1\fR
.fi
.in -2
.sp

.sp
.LP
Create two VNICs with names \fBhello0\fR and \fBtest1\fR on the etherstub. This
operation implicitly creates a virtual switch connecting \fBhello0\fR and
\fBtest1\fR.

.sp
.in +2
.nf
# \fBdladm create-vnic -l stub1 hello0\fR
# \fBdladm create-vnic -l stub1 test1\fR
.fi
.in -2
.sp

.LP
\fBExample 17 \fRShowing Network Usage
.sp
.LP
Network usage statistics can be stored using the extended accounting facility,
\fBacctadm\fR(1M).

.sp
.in +2
.nf
# \fBacctadm -e basic -f /var/log/net.log net\fR
# \fBacctadm net\fR
          Network accounting: active
     Network accounting file: /var/log/net.log
   Tracked Network resources: basic
 Untracked Network resources: src_ip,dst_ip,src_port,dst_port,protocol,
                              dsfield
.fi
.in -2
.sp

.sp
.LP
The saved historical data can be retrieved in summary form using the
\fBshow-usage\fR subcommand:

.sp
.in +2
.nf
# \fBdladm show-usage -f /var/log/net.log\fR
LINK      DURATION  IPACKETS RBYTES      OPACKETS OBYTES      BANDWIDTH
e1000g0   80        1031     546908      0        0           2.44 Kbps
```

```
.fi
.in -2
.sp

.LP
\fBExample 18 \fRDisplaying Bridge Information
.sp
.LP
The following commands use the \fBshow-bridge\fR subcommand with no and various
options.

.sp
.in +2
.nf
# \fBdladm show-bridge\fR
BRIDGE       PROTECT ADDRESS           PRIORITY DESROOT
foo          stp     32768/8:0:20:bf:f 32768    8192/0:d0:0:76:14:38
bar          stp     32768/8:0:20:e5:8 32768    8192/0:d0:0:76:14:38

# \fBdladm show-bridge -l foo\fR
LINK         STATE        UPTIME   DESROOT
hme0         forwarding   117      8192/0:d0:0:76:14:38
qfe1         forwarding   117      8192/0:d0:0:76:14:38

# \fBdladm show-bridge -s foo\fR
BRIDGE       DROPS        FORWARDS
foo          0            302

# \fBdladm show-bridge -ls foo\fR
LINK         DROPS     RECV      XMIT
hme0         0         360832    31797
qfe1         0         322311    356852

# \fBdladm show-bridge -f foo\fR
DEST              AGE     FLAGS  OUTPUT
8:0:20:bc:a7:dc   10.860  --     hme0
8:0:20:bf:f9:69   --      L      hme0
8:0:20:c0:20:26   17.420  --     hme0
8:0:20:e5:86:11   --      L      qfe1
.fi
.in -2
.sp

.LP
\fBExample 19 \fRCreating an IPv4 Tunnel
.sp
.LP
The following sequence of commands creates and then displays a persistent IPv4
tunnel link named \fBmytunnel0\fR between 66.1.2.3 and 192.4.5.6:

.sp
.in +2
.nf
# \fBdladm create-iptun -T ipv4 -s 66.1.2.3 -d 192.4.5.6 mytunnel0\fR
# \fBdladm show-iptun mytunnel0\fR
LINK            TYPE  FLAGS  SOURCE              DESTINATION
mytunnel0       ipv4  --     66.1.2.3            192.4.5.6
.fi
.in -2
.sp

.sp
.LP
A point-to-point IP interface can then be created over this tunnel link:

.sp
```



```
.in +2
.nf
# \fBifconfig mytunnel0 plumb 10.1.0.1 10.1.0.2 up\fR
.fi
.in -2
.sp

.sp
.LP
As with any other IP interface, configuration persistence for this IP interface
is achieved by placing the desired \fBifconfig\fR commands (in this case, the
command for "\fB10.1.0.1 10.1.0.2\fR") into \fB/etc/hostname.mytunnel0\fR.

.LP
\fBExample 20 \fRCreating a 6to4 Tunnel
.sp
.LP
The following command creates a 6to4 tunnel link. The IPv4 address of the 6to4
router is 75.10.11.12.

.sp
.in +2
.nf
# \fBdladm create-iptun -T 6to4 -s 75.10.11.12 sitetunnel0\fR
# \fBdladm show-iptun sitetunnel0\fR
LINK            TYPE  FLAGS  SOURCE              DESTINATION
sitetunnel0     6to4  --     75.10.11.12         --
.fi
.in -2
.sp

.sp
.LP
The following command plumbs an IPv6 interface on this tunnel:

.sp
.in +2
.nf
# \fBifconfig sitetunnel0 inet6 plumb up\fR
# \fBifconfig sitetunnel0 inet6\fR
sitetunnel0: flags=2200041 <UP,RUNNING,NONUD,IPv6> mtu 65515 index 3
        inet tunnel src 75.10.11.12
        tunnel hop limit 64
        inet6 2002:4b0a:b0c::1/16
.fi
.in -2
.sp

.sp
.LP
Note that the system automatically configures the IPv6 address on the 6to4 IP
interface. See \fBifconfig\fR(1M) for a description of how IPv6 addresses are
configured on 6to4 tunnel links.

.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the following attributes:
.sp
.LP
\fB/usr/sbin\fR
.sp

.sp
.TS
box;
```

```
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Committed
.TE

.sp
.LP
\fB/sbin\fR
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Committed
.TE

.SH SEE ALSO
.sp
.LP
\fBacctadm\fR(1M), \fBautopush\fR(1M), \fBifconfig\fR(1M), \fBipsecconf\fR(1M),
\fBndd\fR(1M), \fBpsrset\fR(1M), \fBwpad\fR(1M), \fBzonecfg\fR(1M),
\fBattributes\fR(5), \fBieee802.3\fR(5), \fBdlpi\fR(7P)
.SH NOTES
.sp
.LP
The preferred method of referring to an aggregation in the aggregation
subcommands is by its link name. Referring to an aggregation by its integer
\fIkey\fR is supported for backward compatibility, but is not necessary. When
creating an aggregation, if a \fIkey\fR is specified instead of a link name,
the aggregation's link name will be automatically generated by \fBdladm\fR as
\fBaggr\fR\fIkey\fR.
```