```
**********************************************************
  121858 Thu Feb 20 18:59:04 2014
new/usr/src/lib/libdladm/common/linkprop.c
2553 mac address should be a dladm link property
**********************************************************
______unchanged_portion_omitted_

 129 static dld_ioc_macprop_t *i_dladm_buf_alloc_by_name(size_t, datalink_id_t,
 130                         const char *, uint_t, dladm_status_t *);
 131 static dld_ioc_macprop_t *i_dladm_buf_alloc_by_id(size_t, datalink_id_t,
 132                         mac_prop_id_t, uint_t, dladm_status_t *);
 133 static dladm_status_t   i_dladm_get_public_prop(dladm_handle_t, datalink_id_t,
 134                         char *, uint_t, uint_t *, void *, size_t);

 136 static dladm_status_t   i_dladm_set_private_prop(dladm_handle_t, datalink_id_t,
 137                         const char *, char **, uint_t, uint_t);
 138 static dladm_status_t   i_dladm_get_priv_prop(dladm_handle_t, datalink_id_t,
 139                         const char *, char **, uint_t *, dladm_prop_type_t,
 140                         uint_t);
 141 static dladm_status_t   i_dladm_macprop(dladm_handle_t, void *, boolean_t);
 142 static const char       *dladm_perm2str(uint_t, char *);
 143 static link_attr_t      *dladm_name2prop(const char *);
 144 static link_attr_t      *dladm_id2prop(mac_prop_id_t);

 146 static pd_getf_t        get_zone, get_autopush, get_rate_mod, get_rate,
 147                         get_speed, get_channel, get_powermode, get_radio,
 148                         get_duplex, get_link_state, get_binary, get_uint32,
 149                         get_flowctl, get_maxbw, get_cpus, get_priority,
 150                         get_tagmode, get_range, get_stp, get_bridge_forward,
 151                         get_bridge_pvid, get_protection, get_rxrings,
 152                         get_txrings, get_cntavail, get_macaddr,
 152                         get_txrings, get_cntavail,
 153                         get_allowedips, get_allowedcids, get_pool,
 154                         get_rings_range, get_linkmode_prop;

 156 static pd_setf_t        set_zone, set_rate, set_powermode, set_radio,
 157                         set_public_prop, set_resource, set_stp_prop,
 158                         set_bridge_forward, set_bridge_pvid, set_macaddr;
 158                         set_bridge_forward, set_bridge_pvid;

 160 static pd_checkf_t      check_zone, check_autopush, check_rate, check_hoplimit,
 161                         check_encaplim, check_uint32, check_maxbw, check_cpus,
 162                         check_stp_prop, check_bridge_pvid, check_allowedips,
 163                         check_allowedcids, check_rings, check_macaddr,
 163                         check_allowedcids, check_rings,
 164                         check_pool, check_prop;

 166 struct prop_desc {
 167         /*
 168          * link property name
 169          */
 170         char                    *pd_name;

 172         /*
 173          * default property value, can be set to { "", NULL }
 174          */
 175         val_desc_t              pd_defval;

 177         /*
 178          * list of optional property values, can be NULL.
 179          *
 180          * This is set to non-NULL if there is a list of possible property
 181          * values.  pd_optval would point to the array of possible values.
 182          */
 183         val_desc_t              *pd_optval;
```



```
 185         /*
 186          * count of the above optional property values. 0 if pd_optval is NULL.
 187          */
 188         uint_t                  pd_noptval;

 190         /*
 191          * callback to set link property; set to NULL if this property is
 192          * read-only and may be called before or after permanent update; see
 193          * flags.
 194          */
 195         pd_setf_t               *pd_set;

 197         /*
 198          * callback to get modifiable link property
 199          */
 200         pd_getf_t               *pd_getmod;

 202         /*
 203          * callback to get current link property
 204          */
 205         pd_getf_t               *pd_get;

 207         /*
 208          * callback to validate link property value, set to NULL if pd_optval
 209          * is not NULL. In that case, validate the value by comparing it with
 210          * the pd_optval. Return a val_desc_t array pointer if the value is
 211          * valid.
 212          */
 213         pd_checkf_t             *pd_check;

 215         uint_t                  pd_flags;
 216 #define PD_TEMPONLY     0x1     /* property is temporary only */
 217 #define PD_CHECK_ALLOC  0x2     /* alloc vd_val as part of pd_check */
 218 #define PD_AFTER_PERM   0x4     /* pd_set after db update; no temporary */
 219         /*
 220          * indicate link classes this property applies to.
 221          */
 222         datalink_class_t        pd_class;

 224         /*
 225          * indicate link media type this property applies to.
 226          */
 227         datalink_media_t        pd_dmedia;
 228 };

 230 #define MAC_PROP_BUFSIZE(v)     sizeof (dld_ioc_macprop_t) + (v) - 1

 232 /*
 233  * Supported link properties enumerated in the prop_table[] array are
 234  * computed using the callback functions in that array. To compute the
 235  * property value, multiple distinct system calls may be needed (e.g.,
 236  * for wifi speed, we need to issue system calls to get desired/supported
 237  * rates). The link_attr[] table enumerates the interfaces to the kernel,
 238  * and the type/size of the data passed in the user-kernel interface.
 239  */
 240 static link_attr_t link_attr[] = {
 241         { MAC_PROP_DUPLEX,      sizeof (link_duplex_t), "duplex"},

 243         { MAC_PROP_SPEED,       sizeof (uint64_t),      "speed"},

 245         { MAC_PROP_STATUS,      sizeof (link_state_t),  "state"},

 247         { MAC_PROP_AUTONEG,     sizeof (uint8_t),       "adv_autoneg_cap"},

 249         { MAC_PROP_MTU,         sizeof (uint32_t),      "mtu"},
```

```
251         { MAC_PROP_FLOWCTRL,    sizeof (link_flowctrl_t), "flowctrl"},
253         { MAC_PROP_ZONE,        sizeof (dld_ioc_zid_t), "zone"},
255         { MAC_PROP_AUTOPUSH,    sizeof (struct dlautopush), "autopush"},
257         { MAC_PROP_ADV_10GFDX_CAP, sizeof (uint8_t),    "adv_10gfdx_cap"},
259         { MAC_PROP_EN_10GFDX_CAP, sizeof (uint8_t),     "en_10gfdx_cap"},
261         { MAC_PROP_ADV_1000FDX_CAP, sizeof (uint8_t),   "adv_1000fdx_cap"},
263         { MAC_PROP_EN_1000FDX_CAP, sizeof (uint8_t),    "en_1000fdx_cap"},
265         { MAC_PROP_ADV_1000HDX_CAP, sizeof (uint8_t),   "adv_1000hdx_cap"},
267         { MAC_PROP_EN_1000HDX_CAP, sizeof (uint8_t),    "en_1000hdx_cap"},
269         { MAC_PROP_ADV_100FDX_CAP, sizeof (uint8_t),    "adv_100fdx_cap"},
271         { MAC_PROP_EN_100FDX_CAP, sizeof (uint8_t),     "en_100fdx_cap"},
273         { MAC_PROP_ADV_100HDX_CAP, sizeof (uint8_t),    "adv_100hdx_cap"},
275         { MAC_PROP_EN_100HDX_CAP, sizeof (uint8_t),     "en_100hdx_cap"},
277         { MAC_PROP_ADV_10FDX_CAP, sizeof (uint8_t),     "adv_10fdx_cap"},
279         { MAC_PROP_EN_10FDX_CAP, sizeof (uint8_t),      "en_10fdx_cap"},
281         { MAC_PROP_ADV_10HDX_CAP, sizeof (uint8_t),     "adv_10hdx_cap"},
283         { MAC_PROP_EN_10HDX_CAP, sizeof (uint8_t),      "en_10hdx_cap"},
285         { MAC_PROP_WL_ESSID,    sizeof (wl_linkstatus_t), "essid"},
287         { MAC_PROP_WL_BSSID,    sizeof (wl_bssid_t),    "bssid"},
289         { MAC_PROP_WL_BSSTYPE,  sizeof (wl_bss_type_t), "bsstype"},
291         { MAC_PROP_WL_LINKSTATUS, sizeof (wl_linkstatus_t), "wl_linkstatus"},
293         /* wl_rates_t has variable length */
294         { MAC_PROP_WL_DESIRED_RATES, sizeof (wl_rates_t), "desired_rates"},
296         /* wl_rates_t has variable length */
297         { MAC_PROP_WL_SUPPORTED_RATES, sizeof (wl_rates_t), "supported_rates"},
299         { MAC_PROP_WL_AUTH_MODE, sizeof (wl_authmode_t), "authmode"},
301         { MAC_PROP_WL_ENCRYPTION, sizeof (wl_encryption_t), "encryption"},
303         { MAC_PROP_WL_RSSI,     sizeof (wl_rssi_t),     "signal"},
305         { MAC_PROP_WL_PHY_CONFIG, sizeof (wl_phy_conf_t), "phy_conf"},
307         { MAC_PROP_WL_CAPABILITY, sizeof (wl_capability_t), "capability"},
309         { MAC_PROP_WL_WPA,      sizeof (wl_wpa_t),      "wpa"},
311         /*  wl_wpa_ess_t has variable length */
312         { MAC_PROP_WL_SCANRESULTS, sizeof (wl_wpa_ess_t), "scan_results"},
314         { MAC_PROP_WL_POWER_MODE, sizeof (wl_ps_mode_t), "powermode"},
316         { MAC_PROP_WL_RADIO,    sizeof (dladm_wlan_radio_t), "wl_radio"},
```



```
318         { MAC_PROP_WL_ESS_LIST, sizeof (wl_ess_list_t), "wl_ess_list"},
320         { MAC_PROP_WL_KEY_TAB,  sizeof (wl_wep_key_tab_t), "wl_wep_key"},
322         { MAC_PROP_WL_CREATE_IBSS, sizeof (wl_create_ibss_t), "createibss"},
324         /* wl_wpa_ie_t has variable length */
325         { MAC_PROP_WL_SETOPTIE, sizeof (wl_wpa_ie_t),   "set_ie"},
327         { MAC_PROP_WL_DELKEY,   sizeof (wl_del_key_t),  "wpa_del_key"},
329         { MAC_PROP_WL_KEY,      sizeof (wl_key_t),      "wl_key"},
331         { MAC_PROP_WL_MLME,     sizeof (wl_mlme_t),     "mlme"},
333         { MAC_PROP_TAGMODE,     sizeof (link_tagmode_t),        "tagmode"},
335         { MAC_PROP_IPTUN_HOPLIMIT, sizeof (uint32_t),   "hoplimit"},
337         { MAC_PROP_IPTUN_ENCAPLIMIT, sizeof (uint32_t), "encaplimit"},
339         { MAC_PROP_PVID,        sizeof (uint16_t),      "default_tag"},
341         { MAC_PROP_LLIMIT,      sizeof (uint32_t),      "learn_limit"},
343         { MAC_PROP_LDECAY,      sizeof (uint32_t),      "learn_decay"},
345         { MAC_PROP_RESOURCE,    sizeof (mac_resource_props_t),  "resource"},
347         { MAC_PROP_RESOURCE_EFF, sizeof (mac_resource_props_t),
348           "resource-effective"},
350         { MAC_PROP_RXRINGSRANGE, sizeof (mac_propval_range_t),  "rxrings"},
352         { MAC_PROP_TXRINGSRANGE, sizeof (mac_propval_range_t),  "txrings"},
354         { MAC_PROP_MAX_TX_RINGS_AVAIL,  sizeof (uint_t),
355           "txrings-available"},
357         { MAC_PROP_MAX_RX_RINGS_AVAIL,  sizeof (uint_t),
358           "rxrings-available"},
360         { MAC_PROP_MAX_RXHWCLNT_AVAIL,  sizeof (uint_t), "rxhwclnt-available"},
362         { MAC_PROP_MAX_TXHWCLNT_AVAIL,  sizeof (uint_t), "txhwclnt-available"},
364         { MAC_PROP_IB_LINKMODE, sizeof (uint32_t),      "linkmode"},
366         { MAC_PROP_MACADDRESS,  sizeof (mac_addrprop_t), "mac-address"},
368 #endif /* ! codereview */
369         { MAC_PROP_PRIVATE,     0,                      "driver-private"}
370 };
372 typedef struct bridge_public_prop_s {
373         const char      *bpp_name;
374         int             bpp_code;
375 } bridge_public_prop_t;
377 static const bridge_public_prop_t bridge_prop[] = {
378         { "stp", PT_CFG_NON_STP },
379         { "stp_priority", PT_CFG_PRIO },
380         { "stp_cost", PT_CFG_COST },
381         { "stp_edge", PT_CFG_EDGE },
382         { "stp_p2p", PT_CFG_P2P },
```

```
383         { "stp_mcheck", PT_CFG_MCHECK },
384         { NULL, 0 }
385 };

387 static  val_desc_t      link_duplex_vals[] = {
388         { "half",       LINK_DUPLEX_HALF        },
389         { "full",       LINK_DUPLEX_HALF        }
390 };
391 static  val_desc_t      link_status_vals[] = {
392         { "up",         LINK_STATE_UP           },
393         { "down",       LINK_STATE_DOWN         }
394 };
395 static  val_desc_t      link_01_vals[] = {
396         { "1",          1                       },
397         { "0",          0                       }
398 };
399 static  val_desc_t      link_flow_vals[] = {
400         { "no",         LINK_FLOWCTRL_NONE      },
401         { "tx",         LINK_FLOWCTRL_TX        },
402         { "rx",         LINK_FLOWCTRL_RX        },
403         { "bi",         LINK_FLOWCTRL_BI        }
404 };
405 static  val_desc_t      link_priority_vals[] = {
406         { "low",        MPL_LOW },
407         { "medium",     MPL_MEDIUM      },
408         { "high",       MPL_HIGH        }
409 };

411 static val_desc_t       link_tagmode_vals[] = {
412         { "normal",     LINK_TAGMODE_NORMAL     },
413         { "vlanonly",   LINK_TAGMODE_VLANONLY   }
414 };

416 static  val_desc_t      link_protect_vals[] = {
417         { "mac-nospoof",        MPT_MACNOSPOOF  },
418         { "restricted",         MPT_RESTRICTED  },
419         { "ip-nospoof",         MPT_IPNOSPOOF   },
420         { "dhcp-nospoof",       MPT_DHCPNOSPOOF },
421 };

423 static val_desc_t       dladm_wlan_radio_vals[] = {
424         { "on",         DLADM_WLAN_RADIO_ON     },
425         { "off",        DLADM_WLAN_RADIO_OFF    }
426 };

428 static val_desc_t       dladm_wlan_powermode_vals[] = {
429         { "off",        DLADM_WLAN_PM_OFF       },
430         { "fast",       DLADM_WLAN_PM_FAST      },
431         { "max",        DLADM_WLAN_PM_MAX       }
432 };

434 static  val_desc_t      stp_p2p_vals[] = {
435         { "true",       P2P_FORCE_TRUE          },
436         { "false",      P2P_FORCE_FALSE         },
437         { "auto",       P2P_AUTO                }
438 };

440 static  val_desc_t      dladm_part_linkmode_vals[] = {
441         { "cm",         DLADM_PART_CM_MODE      },
442         { "ud",         DLADM_PART_UD_MODE      },
443 };

445 #define VALCNT(vals)    (sizeof ((vals)) / sizeof (val_desc_t))
446 #define RESET_VAL       ((uintptr_t)-1)
447 #define UNSPEC_VAL      ((uintptr_t)-2)
```



```
449 static prop_desc_t      prop_table[] = {
450         { "channel",    { NULL, 0 },
451             NULL, 0, NULL, NULL,
452             get_channel, NULL, 0,
453             DATALINK_CLASS_PHYS, DL_WIFI },

455         { "powermode",  { "off", DLADM_WLAN_PM_OFF },
456             dladm_wlan_powermode_vals, VALCNT(dladm_wlan_powermode_vals),
457             set_powermode, NULL,
458             get_powermode, NULL, 0,
459             DATALINK_CLASS_PHYS, DL_WIFI },

461         { "radio",      { "on", DLADM_WLAN_RADIO_ON },
462             dladm_wlan_radio_vals, VALCNT(dladm_wlan_radio_vals),
463             set_radio, NULL,
464             get_radio, NULL, 0,
465             DATALINK_CLASS_PHYS, DL_WIFI },

467         { "linkmode",   { "cm", DLADM_PART_CM_MODE },
468             dladm_part_linkmode_vals, VALCNT(dladm_part_linkmode_vals),
469             set_public_prop, NULL, get_linkmode_prop, NULL, 0,
470             DATALINK_CLASS_PART, DL_IB },

472         { "speed",      { "", 0 }, NULL, 0,
473             set_rate, get_rate_mod,
474             get_rate, check_rate, 0,
475             DATALINK_CLASS_PHYS, DATALINK_ANY_MEDIATYPE },

477         { "autopush",   { "", 0 }, NULL, 0,
478             set_public_prop, NULL,
479             get_autopush, check_autopush, PD_CHECK_ALLOC,
480             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

482         { "zone",       { "", 0 }, NULL, 0,
483             set_zone, NULL,
484             get_zone, check_zone, PD_TEMPONLY|PD_CHECK_ALLOC,
485             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

487         { "duplex",     { "", 0 },
488             link_duplex_vals, VALCNT(link_duplex_vals),
489             NULL, NULL, get_duplex, NULL,
490             0, DATALINK_CLASS_PHYS, DL_ETHER },

492         { "state",      { "up", LINK_STATE_UP },
493             link_status_vals, VALCNT(link_status_vals),
494             NULL, NULL, get_link_state, NULL,
495             0, DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

497         { "adv_autoneg_cap", { "", 0 },
498             link_01_vals, VALCNT(link_01_vals),
499             set_public_prop, NULL, get_binary, NULL,
500             0, DATALINK_CLASS_PHYS, DL_ETHER },

502         { "mtu", { "", 0 }, NULL, 0,
503             set_public_prop, get_range,
504             get_uint32, check_uint32, 0, DATALINK_CLASS_ALL,
505             DATALINK_ANY_MEDIATYPE },

507         { "flowctrl", { "", 0 },
508             link_flow_vals, VALCNT(link_flow_vals),
509             set_public_prop, NULL, get_flowctl, NULL,
510             0, DATALINK_CLASS_PHYS, DL_ETHER },

512         { "adv_10gfdx_cap", { "", 0 },
513             link_01_vals, VALCNT(link_01_vals),
514             NULL, NULL, get_binary, NULL,
```

```
515             0, DATALINK_CLASS_PHYS, DL_ETHER },

517         { "en_10gfdx_cap", { "", 0 },
518             link_01_vals, VALCNT(link_01_vals),
519             set_public_prop, NULL, get_binary, NULL,
520             0, DATALINK_CLASS_PHYS, DL_ETHER },

522         { "adv_1000fdx_cap", { "", 0 },
523             link_01_vals, VALCNT(link_01_vals),
524             NULL, NULL, get_binary, NULL,
525             0, DATALINK_CLASS_PHYS, DL_ETHER },

527         { "en_1000fdx_cap", { "", 0 },
528             link_01_vals, VALCNT(link_01_vals),
529             set_public_prop, NULL, get_binary, NULL,
530             0, DATALINK_CLASS_PHYS, DL_ETHER },

532         { "adv_1000hdx_cap", { "", 0 },
533             link_01_vals, VALCNT(link_01_vals),
534             NULL, NULL, get_binary, NULL,
535             0, DATALINK_CLASS_PHYS, DL_ETHER },

537         { "en_1000hdx_cap", { "", 0 },
538             link_01_vals, VALCNT(link_01_vals),
539             set_public_prop, NULL, get_binary, NULL,
540             0, DATALINK_CLASS_PHYS, DL_ETHER },

542         { "adv_100fdx_cap", { "", 0 },
543             link_01_vals, VALCNT(link_01_vals),
544             NULL, NULL, get_binary, NULL,
545             0, DATALINK_CLASS_PHYS, DL_ETHER },

547         { "en_100fdx_cap", { "", 0 },
548             link_01_vals, VALCNT(link_01_vals),
549             set_public_prop, NULL, get_binary, NULL,
550             0, DATALINK_CLASS_PHYS, DL_ETHER },

552         { "adv_100hdx_cap", { "", 0 },
553             link_01_vals, VALCNT(link_01_vals),
554             NULL, NULL, get_binary, NULL,
555             0, DATALINK_CLASS_PHYS, DL_ETHER },

557         { "en_100hdx_cap", { "", 0 },
558             link_01_vals, VALCNT(link_01_vals),
559             set_public_prop, NULL, get_binary, NULL,
560             0, DATALINK_CLASS_PHYS, DL_ETHER },

562         { "adv_10fdx_cap", { "", 0 },
563             link_01_vals, VALCNT(link_01_vals),
564             NULL, NULL, get_binary, NULL,
565             0, DATALINK_CLASS_PHYS, DL_ETHER },

567         { "en_10fdx_cap", { "", 0 },
568             link_01_vals, VALCNT(link_01_vals),
569             set_public_prop, NULL, get_binary, NULL,
570             0, DATALINK_CLASS_PHYS, DL_ETHER },

572         { "adv_10hdx_cap", { "", 0 },
573             link_01_vals, VALCNT(link_01_vals),
574             NULL, NULL, get_binary, NULL,
575             0, DATALINK_CLASS_PHYS, DL_ETHER },

577         { "en_10hdx_cap", { "", 0 },
578             link_01_vals, VALCNT(link_01_vals),
579             set_public_prop, NULL, get_binary, NULL,
580             0, DATALINK_CLASS_PHYS, DL_ETHER },
```



```
582         { "maxbw", { "--", RESET_VAL }, NULL, 0,
583             set_resource, NULL,
584             get_maxbw, check_maxbw, PD_CHECK_ALLOC,
585             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

587         { "cpus", { "--", RESET_VAL }, NULL, 0,
588             set_resource, NULL,
589             get_cpus, check_cpus, 0,
590             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

592         { "cpus-effective", { "--", 0 },
593             NULL, 0, NULL, NULL,
594             get_cpus, 0, 0,
595             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

597         { "pool", { "--", RESET_VAL }, NULL, 0,
598             set_resource, NULL,
599             get_pool, check_pool, 0,
600             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

602         { "pool-effective", { "--", 0 },
603             NULL, 0, NULL, NULL,
604             get_pool, 0, 0,
605             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

607         { "priority", { "high", MPL_RESET },
608             link_priority_vals, VALCNT(link_priority_vals), set_resource,
609             NULL, get_priority, check_prop, 0,
610             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

612         { "tagmode", { "vlanonly", LINK_TAGMODE_VLANONLY },
613             link_tagmode_vals, VALCNT(link_tagmode_vals),
614             set_public_prop, NULL, get_tagmode,
615             NULL, 0,
616             DATALINK_CLASS_PHYS | DATALINK_CLASS_AGGR | DATALINK_CLASS_VNIC,
617             DL_ETHER },

619         { "hoplimit", { "", 0 }, NULL, 0,
620             set_public_prop, get_range, get_uint32,
621             check_hoplimit, 0, DATALINK_CLASS_IPTUN, DATALINK_ANY_MEDIATYPE},

623         { "encaplimit", { "", 0 }, NULL, 0,
624             set_public_prop, get_range, get_uint32,
625             check_encaplim, 0, DATALINK_CLASS_IPTUN, DL_IPV6},

627         { "forward", { "1", 1 },
628             link_01_vals, VALCNT(link_01_vals),
629             set_bridge_forward, NULL, get_bridge_forward, NULL, PD_AFTER_PERM,
630             DATALINK_CLASS_ALL & ~DATALINK_CLASS_VNIC, DL_ETHER },

632         { "default_tag", { "1", 1 }, NULL, 0,
633             set_bridge_pvid, NULL, get_bridge_pvid, check_bridge_pvid,
634             0, DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
635             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

637         { "learn_limit", { "1000", 1000 }, NULL, 0,
638             set_public_prop, NULL, get_uint32,
639             check_uint32, 0,
640             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
641             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

643         { "learn_decay", { "200", 200 }, NULL, 0,
644             set_public_prop, NULL, get_uint32,
645             check_uint32, 0,
646             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
```

```
647             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

649         { "stp", { "1", 1 },
650             link_01_vals, VALCNT(link_01_vals),
651             set_stp_prop, NULL, get_stp, NULL, PD_AFTER_PERM,
652             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
653             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

655         { "stp_priority", { "128", 128 }, NULL, 0,
656             set_stp_prop, NULL, get_stp, check_stp_prop, PD_AFTER_PERM,
657             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
658             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

660         { "stp_cost", { "auto", 0 }, NULL, 0,
661             set_stp_prop, NULL, get_stp, check_stp_prop, PD_AFTER_PERM,
662             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
663             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

665         { "stp_edge", { "1", 1 },
666             link_01_vals, VALCNT(link_01_vals),
667             set_stp_prop, NULL, get_stp, NULL, PD_AFTER_PERM,
668             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
669             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

671         { "stp_p2p", { "auto", P2P_AUTO },
672             stp_p2p_vals, VALCNT(stp_p2p_vals),
673             set_stp_prop, NULL, get_stp, NULL, PD_AFTER_PERM,
674             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
675             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

677         { "stp_mcheck", { "0", 0 },
678             link_01_vals, VALCNT(link_01_vals),
679             set_stp_prop, NULL, get_stp, check_stp_prop, PD_AFTER_PERM,
680             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|
681             DATALINK_CLASS_ETHERSTUB|DATALINK_CLASS_SIMNET, DL_ETHER },

683         { "protection", { "--", RESET_VAL },
684             link_protect_vals, VALCNT(link_protect_vals),
685             set_resource, NULL, get_protection, check_prop, 0,
686             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

688         { "mac-address", { "", 0 }, NULL, 0,
689             set_macaddr, NULL, get_macaddr, check_macaddr, PD_CHECK_ALLOC,
690             DATALINK_CLASS_PHYS|DATALINK_CLASS_AGGR|DATALINK_CLASS_VNIC|
691             DATALINK_CLASS_SIMNET, DATALINK_ANY_MEDIATYPE },

693 #endif /* ! codereview */
694         { "allowed-ips", { "--", 0 },
695             NULL, 0, set_resource, NULL,
696             get_allowedips, check_allowedips, PD_CHECK_ALLOC,
697             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

699         { "allowed-dhcp-cids", { "--", 0 },
700             NULL, 0, set_resource, NULL,
701             get_allowedcids, check_allowedcids, PD_CHECK_ALLOC,
702             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

704         { "rxrings", { "--", RESET_VAL }, NULL, 0,
705             set_resource, get_rings_range, get_rxrings, check_rings, 0,
706             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

708         { "rxrings-effective", { "--", 0 },
709             NULL, 0, NULL, NULL,
710             get_rxrings, NULL, 0,
711             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },
```



```
713         { "txrings", { "--", RESET_VAL }, NULL, 0,
714             set_resource, get_rings_range, get_txrings, check_rings, 0,
715             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

717         { "txrings-effective", { "--", 0 },
718             NULL, 0, NULL, NULL,
719             get_txrings, NULL, 0,
720             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

722         { "txrings-available", { "", 0 }, NULL, 0,
723             NULL, NULL, get_cntavail, NULL, 0,
724             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

726         { "rxrings-available", { "", 0 }, NULL, 0,
727             NULL, NULL, get_cntavail, NULL, 0,
728             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

730         { "rxhwclnt-available", { "", 0 }, NULL, 0,
731             NULL, NULL, get_cntavail, NULL, 0,
732             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

734         { "txhwclnt-available", { "", 0 }, NULL, 0,
735             NULL, NULL, get_cntavail, NULL, 0,
736             DATALINK_CLASS_ALL, DATALINK_ANY_MEDIATYPE },

737 };
```

*unchanged_portion_omitted_*

```
2254 /* ARGSUSED */
2255 static dladm_status_t
2256 get_macaddr(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
2257     char **prop_val, uint_t *val_cnt, datalink_media_t media, uint_t flags,
2258     uint_t *perm_flags)
2259 {
2260         mac_addrprop_t  addrprop;
2261         dladm_status_t  status;

2263         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name,
2264             flags, perm_flags, &addrprop, sizeof (addrprop));
2265         if (status != DLADM_STATUS_OK)
2266                 return (status);

2268         (void) _link_ntoa(addrprop.ma_addr, prop_val[0], addrprop.ma_len,
2269             IFT_OTHER);

2271         *val_cnt = 1;

2273         return (DLADM_STATUS_OK);
2274 }

2276 /* ARGSUSED */
2277 static dladm_status_t
2278 set_macaddr(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
2279     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
2280 {
2281         if (val_cnt != 1)
2282                 return (DLADM_STATUS_BADVALCNT);

2284         return (set_public_prop(handle, pdp, linkid, vdp, val_cnt,
2285             flags, media));
2286 }

2288 /* ARGSUSED */
2289 static dladm_status_t
2290 check_macaddr(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
2291     char **prop_val, uint_t *val_cntp, uint_t flags, val_desc_t **vdpp,
```

```
2292     datalink_media_t media)
2293 {
2294         mac_addrprop_t  *addrprop;
2295         val_desc_t      *vdp = *vdpp;
2296         uint_t          val_cnt = *val_cntp;
2297         uchar_t         *macaddr;
2298         int             maclen;
2299         dladm_status_t  status = DLADM_STATUS_OK;

2301         if (val_cnt != 1)
2302                 return (DLADM_STATUS_BADVALCNT);

2304         macaddr = _link_aton(*prop_val, &maclen);
2305         if (macaddr == NULL)
2306                 return (DLADM_STATUS_BADVAL);

2308         addrprop = malloc(sizeof (mac_addrprop_t));
2309         if (addrprop == NULL) {
2310                 status = DLADM_STATUS_NOMEM;
2311                 goto out;
2312         }

2314         (void) memcpy(addrprop->ma_addr, macaddr, maclen);
2315         addrprop->ma_len = maclen;

2317         vdp->vd_val = (uintptr_t)addrprop;

2319 out:
2320         free(macaddr);
2321         return (status);
2322 }

2324 /* ARGSUSED */
2325 static dladm_status_t
2326 #endif /* ! codereview */
2327 set_resource(dladm_handle_t handle, prop_desc_t *pdp,
2328     datalink_id_t linkid, val_desc_t *vdp, uint_t val_cnt,
2329     uint_t flags, datalink_media_t media)
2330 {
2331         mac_resource_props_t    mrp;
2332         dladm_status_t          status = DLADM_STATUS_OK;
2333         dld_ioc_macprop_t       *dip;
2334         int                     i;

2336         bzero(&mrp, sizeof (mac_resource_props_t));
2337         dip = i_dladm_buf_alloc_by_name(0, linkid, "resource",
2338             flags, &status);

2340         if (dip == NULL)
2341                 return (status);

2343         for (i = 0; i < DLADM_MAX_RSRC_PROP; i++) {
2344                 resource_prop_t *rp = &rsrc_prop_table[i];

2346                 if (strcmp(pdp->pd_name, rp->rp_name) != 0)
2347                         continue;

2349                 status = rp->rp_extract(vdp, val_cnt, &mrp);
2350                 if (status != DLADM_STATUS_OK)
2351                         goto done;

2353                 break;
2354         }

2356         (void) memcpy(dip->pr_val, &mrp, dip->pr_valsize);
2357         status = i_dladm_macprop(handle, dip, B_TRUE);
```



```
2359 done:
2360         free(dip);
2361         return (status);
2362 }

2364 /* ARGSUSED */
2365 static dladm_status_t
2366 get_protection(dladm_handle_t handle, prop_desc_t *pdp,
2367     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
2368     datalink_media_t media, uint_t flags, uint_t *perm_flags)
2369 {
2370         mac_resource_props_t    mrp;
2371         mac_protect_t           *p;
2372         dladm_status_t          status;
2373         uint32_t                i, cnt = 0, setbits[32];

2375         status = i_dladm_get_public_prop(handle, linkid, "resource", flags,
2376             perm_flags, &mrp, sizeof (mrp));
2377         if (status != DLADM_STATUS_OK)
2378                 return (status);

2380         p = &mrp.mrp_protect;
2381         if ((mrp.mrp_mask & MRP_PROTECT) == 0) {
2382                 *val_cnt = 0;
2383                 return (DLADM_STATUS_OK);
2384         }
2385         dladm_find_setbits32(p->mp_types, setbits, &cnt);
2386         if (cnt > *val_cnt)
2387                 return (DLADM_STATUS_BADVALCNT);

2389         for (i = 0; i < cnt; i++)
2390                 (void) dladm_protect2str(setbits[i], prop_val[i]);

2392         *val_cnt = cnt;
2393         return (DLADM_STATUS_OK);
2394 }

2396 /* ARGSUSED */
2397 static dladm_status_t
2398 get_allowedips(dladm_handle_t handle, prop_desc_t *pdp,
2399     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
2400     datalink_media_t media, uint_t flags, uint_t *perm_flags)
2401 {
2402         mac_resource_props_t    mrp;
2403         mac_protect_t           *p;
2404         dladm_status_t          status;
2405         int                     i;

2407         status = i_dladm_get_public_prop(handle, linkid, "resource", flags,
2408             perm_flags, &mrp, sizeof (mrp));
2409         if (status != DLADM_STATUS_OK)
2410                 return (status);

2412         p = &mrp.mrp_protect;
2413         if (p->mp_ipaddrcnt == 0) {
2414                 *val_cnt = 0;
2415                 return (DLADM_STATUS_OK);
2416         }
2417         if (p->mp_ipaddrcnt > *val_cnt)
2418                 return (DLADM_STATUS_BADVALCNT);

2420         for (i = 0; i < p->mp_ipaddrcnt; i++) {
2421                 if (p->mp_ipaddrs[i].ip_version == IPV4_VERSION) {
2422                         ipaddr_t        v4addr;
```

```
2424                        v4addr = V4_PART_OF_V6(p->mp_ipaddrs[i].ip_addr);
2425                        (void) dladm_ipv4addr2str(&v4addr, prop_val[i]);
2426                } else {
2427                        (void) dladm_ipv6addr2str(&p->mp_ipaddrs[i].ip_addr,
2428                            prop_val[i]);
2429                }
2430        }
2431        *val_cnt = p->mp_ipaddrcnt;
2432        return (DLADM_STATUS_OK);
2433 }

2435 dladm_status_t
2436 extract_protection(val_desc_t *vdp, uint_t cnt, void *arg)
2437 {
2438        mac_resource_props_t    *mrp = arg;
2439        uint32_t                types = 0;
2440        int                     i;

2442        for (i = 0; i < cnt; i++)
2443                types |= (uint32_t)vdp[i].vd_val;

2445        mrp->mrp_protect.mp_types = types;
2446        mrp->mrp_mask |= MRP_PROTECT;
2447        return (DLADM_STATUS_OK);
2448 }

2450 dladm_status_t
2451 extract_allowedips(val_desc_t *vdp, uint_t cnt, void *arg)
2452 {
2453        mac_resource_props_t    *mrp = arg;
2454        mac_protect_t           *p = &mrp->mrp_protect;
2455        int                     i;

2457        if (vdp->vd_val == 0) {
2458                cnt = (uint_t)-1;
2459        } else {
2460                for (i = 0; i < cnt; i++) {
2461                        bcopy((void *)vdp[i].vd_val, &p->mp_ipaddrs[i],
2462                            sizeof (mac_ipaddr_t));
2463                }
2464        }
2465        p->mp_ipaddrcnt = cnt;
2466        mrp->mrp_mask |= MRP_PROTECT;
2467        return (DLADM_STATUS_OK);
2468 }

2470 static dladm_status_t
2471 check_single_ip(char *buf, mac_ipaddr_t *addr)
2472 {
2473        dladm_status_t  status;
2474        ipaddr_t        v4addr;
2475        in6_addr_t      v6addr;
2476        boolean_t       isv4 = B_TRUE;

2478        status = dladm_str2ipv4addr(buf, &v4addr);
2479        if (status == DLADM_STATUS_INVALID_IP) {
2480                status = dladm_str2ipv6addr(buf, &v6addr);
2481                if (status == DLADM_STATUS_OK)
2482                        isv4 = B_FALSE;
2483        }
2484        if (status != DLADM_STATUS_OK)
2485                return (status);

2487        if (isv4) {
2488                if (v4addr == INADDR_ANY)
2489                        return (DLADM_STATUS_INVALID_IP);
```



```
2491                IN6_IPADDR_TO_V4MAPPED(v4addr, &addr->ip_addr);
2492                addr->ip_version = IPV4_VERSION;
2493        } else {
2494                if (IN6_IS_ADDR_UNSPECIFIED(&v6addr))
2495                        return (DLADM_STATUS_INVALID_IP);

2497                addr->ip_addr = v6addr;
2498                addr->ip_version = IPV6_VERSION;
2499        }
2500        return (DLADM_STATUS_OK);
2501 }

2503 /* ARGSUSED */
2504 static dladm_status_t
2505 check_allowedips(dladm_handle_t handle, prop_desc_t *pdp,
2506     datalink_id_t linkid, char **prop_val, uint_t *val_cntp, uint_t flags,
2507     val_desc_t **vdpp, datalink_media_t media)
2508 {
2509        dladm_status_t  status;
2510        mac_ipaddr_t    *addr;
2511        int             i;
2512        uint_t          val_cnt = *val_cntp;
2513        val_desc_t      *vdp = *vdpp;

2515        if (val_cnt > MPT_MAXIPADDR)
2516                return (DLADM_STATUS_BADVALCNT);

2518        for (i = 0; i < val_cnt; i++) {
2519                if ((addr = calloc(1, sizeof (mac_ipaddr_t))) == NULL) {
2520                        status = DLADM_STATUS_NOMEM;
2521                        goto fail;
2522                }
2523                vdp[i].vd_val = (uintptr_t)addr;

2525                status = check_single_ip(prop_val[i], addr);
2526                if (status != DLADM_STATUS_OK)
2527                        goto fail;
2528        }
2529        return (DLADM_STATUS_OK);

2531 fail:
2532        for (i = 0; i < val_cnt; i++) {
2533                free((void *)vdp[i].vd_val);
2534                vdp[i].vd_val = NULL;
2535        }
2536        return (status);
2537 }

2539 static void
2540 dladm_cid2str(mac_dhcpcid_t *cid, char *buf)
2541 {
2542        char    tmp_buf[DLADM_STRSIZE];
2543        uint_t  hexlen;

2545        switch (cid->dc_form) {
2546        case CIDFORM_TYPED: {
2547                uint16_t        duidtype, hwtype;
2548                uint32_t        timestamp, ennum;
2549                char            *lladdr;

2551                if (cid->dc_len < sizeof (duidtype))
2552                        goto fail;

2554                bcopy(cid->dc_id, &duidtype, sizeof (duidtype));
2555                duidtype = ntohs(duidtype);
```

```
2556                switch (duidtype) {
2557                case DHCPV6_DUID_LLT: {
2558                        duid_llt_t      llt;

2560                        if (cid->dc_len < sizeof (llt))
2561                                goto fail;

2563                        bcopy(cid->dc_id, &llt, sizeof (llt));
2564                        hwtype = ntohs(llt.dllt_hwtype);
2565                        timestamp = ntohl(llt.dllt_time);
2566                        lladdr = _link_ntoa(cid->dc_id + sizeof (llt),
2567                            NULL, cid->dc_len - sizeof (llt), IFT_OTHER);
2568                        if (lladdr == NULL)
2569                                goto fail;

2571                        (void) snprintf(buf, DLADM_STRSIZE, "%d.%d.%d.%s",
2572                            duidtype, hwtype, timestamp, lladdr);
2573                        free(lladdr);
2574                        break;
2575                }
2576                case DHCPV6_DUID_EN: {
2577                        duid_en_t       en;

2579                        if (cid->dc_len < sizeof (en))
2580                                goto fail;

2582                        bcopy(cid->dc_id, &en, sizeof (en));
2583                        ennum = DHCPV6_GET_ENTNUM(&en);
2584                        hexlen = sizeof (tmp_buf);
2585                        if (octet_to_hexascii(cid->dc_id + sizeof (en),
2586                            cid->dc_len - sizeof (en), tmp_buf, &hexlen) != 0)
2587                                goto fail;

2589                        (void) snprintf(buf, DLADM_STRSIZE, "%d.%d.%s",
2590                            duidtype, ennum, tmp_buf);
2591                        break;
2592                }
2593                case DHCPV6_DUID_LL: {
2594                        duid_ll_t       ll;

2596                        if (cid->dc_len < sizeof (ll))
2597                                goto fail;

2599                        bcopy(cid->dc_id, &ll, sizeof (ll));
2600                        hwtype = ntohs(ll.dll_hwtype);
2601                        lladdr = _link_ntoa(cid->dc_id + sizeof (ll),
2602                            NULL, cid->dc_len - sizeof (ll), IFT_OTHER);
2603                        if (lladdr == NULL)
2604                                goto fail;

2606                        (void) snprintf(buf, DLADM_STRSIZE, "%d.%d.%s",
2607                            duidtype, hwtype, lladdr);
2608                        free(lladdr);
2609                        break;
2610                }
2611                default: {
2612                        hexlen = sizeof (tmp_buf);
2613                        if (octet_to_hexascii(cid->dc_id + sizeof (duidtype),
2614                            cid->dc_len - sizeof (duidtype),
2615                            tmp_buf, &hexlen) != 0)
2616                                goto fail;

2618                        (void) snprintf(buf, DLADM_STRSIZE, "%d.%s",
2619                            duidtype, tmp_buf);
2620                }
2621                }
```



```
2622                        break;
2623                }
2624                case CIDFORM_HEX: {
2625                        hexlen = sizeof (tmp_buf);
2626                        if (octet_to_hexascii(cid->dc_id, cid->dc_len,
2627                            tmp_buf, &hexlen) != 0)
2628                                goto fail;

2630                        (void) snprintf(buf, DLADM_STRSIZE, "0x%s", tmp_buf);
2631                        break;
2632                }
2633                case CIDFORM_STR: {
2634                        int     i;

2636                        for (i = 0; i < cid->dc_len; i++) {
2637                                if (!isprint(cid->dc_id[i]))
2638                                        goto fail;
2639                        }
2640                        (void) snprintf(buf, DLADM_STRSIZE, "%s", cid->dc_id);
2641                        break;
2642                }
2643                default:
2644                        goto fail;
2645                }
2646                return;

2648        fail:
2649                (void) snprintf(buf, DLADM_STRSIZE, "<unknown>");
2650        }

2652        static dladm_status_t
2653        dladm_str2cid(char *buf, mac_dhcpcid_t *cid)
2654        {
2655                char    *ptr = buf;
2656                char    tmp_buf[DLADM_STRSIZE];
2657                uint_t  hexlen, cidlen;

2659                bzero(cid, sizeof (*cid));
2660                if (isdigit(*ptr) &&
2661                    ptr[strspn(ptr, "0123456789")] == '.') {
2662                        char    *cp;
2663                        ulong_t duidtype;
2664                        ulong_t subtype;
2665                        ulong_t timestamp;
2666                        uchar_t *lladdr;
2667                        int     addrlen;

2669                        errno = 0;
2670                        duidtype = strtoul(ptr, &cp, 0);
2671                        if (ptr == cp || errno != 0 || *cp != '.' ||
2672                            duidtype > USHRT_MAX)
2673                                return (DLADM_STATUS_BADARG);
2674                        ptr = cp + 1;

2676                        if (duidtype != 0 && duidtype <= DHCPV6_DUID_LL) {
2677                                errno = 0;
2678                                subtype = strtoul(ptr, &cp, 0);
2679                                if (ptr == cp || errno != 0 || *cp != '.')
2680                                        return (DLADM_STATUS_BADARG);
2681                                ptr = cp + 1;
2682                        }
2683                        switch (duidtype) {
2684                        case DHCPV6_DUID_LLT: {
2685                                duid_llt_t      llt;

2687                                errno = 0;
```

```
2688                        timestamp = strtoul(ptr, &cp, 0);
2689                        if (ptr == cp || errno != 0 || *cp != '.')
2690                                return (DLADM_STATUS_BADARG);

2692                        ptr = cp + 1;
2693                        lladdr = _link_aton(ptr, &addrlen);
2694                        if (lladdr == NULL)
2695                                return (DLADM_STATUS_BADARG);

2697                        cidlen = sizeof (llt) + addrlen;
2698                        if (cidlen > sizeof (cid->dc_id)) {
2699                                free(lladdr);
2700                                return (DLADM_STATUS_TOOSMALL);
2701                        }
2702                        llt.dllt_dutype = htons(duidtype);
2703                        llt.dllt_hwtype = htons(subtype);
2704                        llt.dllt_time = htonl(timestamp);
2705                        bcopy(&llt, cid->dc_id, sizeof (llt));
2706                        bcopy(lladdr, cid->dc_id + sizeof (llt), addrlen);
2707                        free(lladdr);
2708                        break;
2709                }
2710                case DHCPV6_DUID_LL: {
2711                        duid_ll_t       ll;

2713                        lladdr = _link_aton(ptr, &addrlen);
2714                        if (lladdr == NULL)
2715                                return (DLADM_STATUS_BADARG);

2717                        cidlen = sizeof (ll) + addrlen;
2718                        if (cidlen > sizeof (cid->dc_id)) {
2719                                free(lladdr);
2720                                return (DLADM_STATUS_TOOSMALL);
2721                        }
2722                        ll.dll_dutype = htons(duidtype);
2723                        ll.dll_hwtype = htons(subtype);
2724                        bcopy(&ll, cid->dc_id, sizeof (ll));
2725                        bcopy(lladdr, cid->dc_id + sizeof (ll), addrlen);
2726                        free(lladdr);
2727                        break;
2728                }
2729                default: {
2730                        hexlen = sizeof (tmp_buf);
2731                        if (hexascii_to_octet(ptr, strlen(ptr),
2732                            tmp_buf, &hexlen) != 0)
2733                                return (DLADM_STATUS_BADARG);

2735                        if (duidtype == DHCPV6_DUID_EN) {
2736                                duid_en_t       en;

2738                                en.den_dutype = htons(duidtype);
2739                                DHCPV6_SET_ENTNUM(&en, subtype);

2741                                cidlen = sizeof (en) + hexlen;
2742                                if (cidlen > sizeof (cid->dc_id))
2743                                        return (DLADM_STATUS_TOOSMALL);

2745                                bcopy(&en, cid->dc_id, sizeof (en));
2746                                bcopy(tmp_buf, cid->dc_id + sizeof (en),
2747                                    hexlen);
2748                        } else {
2749                                uint16_t        dutype = htons(duidtype);

2751                                cidlen = sizeof (dutype) + hexlen;
2752                                if (cidlen > sizeof (cid->dc_id))
2753                                        return (DLADM_STATUS_TOOSMALL);
```



```
2755                                bcopy(&dutype, cid->dc_id, sizeof (dutype));
2756                                bcopy(tmp_buf, cid->dc_id + sizeof (dutype),
2757                                    hexlen);
2758                        }
2759                        break;
2760                }
2761                }
2762                cid->dc_form = CIDFORM_TYPED;
2763        } else if (strncasecmp("0x", ptr, 2) == 0 && ptr[2] != '\0') {
2764                ptr += 2;
2765                hexlen = sizeof (tmp_buf);
2766                if (hexascii_to_octet(ptr, strlen(ptr), tmp_buf,
2767                    &hexlen) != 0) {
2768                        return (DLADM_STATUS_BADARG);
2769                }
2770                cidlen = hexlen;
2771                if (cidlen > sizeof (cid->dc_id))
2772                        return (DLADM_STATUS_TOOSMALL);

2774                bcopy(tmp_buf, cid->dc_id, cidlen);
2775                cid->dc_form = CIDFORM_HEX;
2776        } else {
2777                cidlen = strlen(ptr);
2778                if (cidlen > sizeof (cid->dc_id))
2779                        return (DLADM_STATUS_TOOSMALL);

2781                bcopy(ptr, cid->dc_id, cidlen);
2782                cid->dc_form = CIDFORM_STR;
2783        }
2784        cid->dc_len = cidlen;
2785        return (DLADM_STATUS_OK);
2786 }

2788 /* ARGSUSED */
2789 static dladm_status_t
2790 get_allowedcids(dladm_handle_t handle, prop_desc_t *pdp,
2791     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
2792     datalink_media_t media, uint_t flags, uint_t *perm_flags)
2793 {
2794        mac_resource_props_t    mrp;
2795        mac_protect_t           *p;
2796        dladm_status_t          status;
2797        int                     i;

2799        status = i_dladm_get_public_prop(handle, linkid, "resource", flags,
2800            perm_flags, &mrp, sizeof (mrp));
2801        if (status != DLADM_STATUS_OK)
2802                return (status);

2804        p = &mrp.mrp_protect;
2805        if (p->mp_cidcnt == 0) {
2806                *val_cnt = 0;
2807                return (DLADM_STATUS_OK);
2808        }
2809        if (p->mp_cidcnt > *val_cnt)
2810                return (DLADM_STATUS_BADVALCNT);

2812        for (i = 0; i < p->mp_cidcnt; i++) {
2813                mac_dhcpcid_t   *cid = &p->mp_cids[i];

2815                dladm_cid2str(cid, prop_val[i]);
2816        }
2817        *val_cnt = p->mp_cidcnt;
2818        return (DLADM_STATUS_OK);
2819 }
```

```
2821 dladm_status_t
2822 extract_allowedcids(val_desc_t *vdp, uint_t cnt, void *arg)
2823 {
2824         mac_resource_props_t    *mrp = arg;
2825         mac_protect_t           *p = &mrp->mrp_protect;
2826         int                     i;
2827
2828         if (vdp->vd_val == 0) {
2829                 cnt = (uint_t)-1;
2830         } else {
2831                 for (i = 0; i < cnt; i++) {
2832                         bcopy((void *)vdp[i].vd_val, &p->mp_cids[i],
2833                             sizeof (mac_dhcpcid_t));
2834                 }
2835         }
2836         p->mp_cidcnt = cnt;
2837         mrp->mrp_mask |= MRP_PROTECT;
2838         return (DLADM_STATUS_OK);
2839 }
2840
2841 /* ARGSUSED */
2842 static dladm_status_t
2843 check_allowedcids(dladm_handle_t handle, prop_desc_t *pdp,
2844     datalink_id_t linkid, char **prop_val, uint_t *val_cntp,
2845     uint_t flags, val_desc_t **vdpp, datalink_media_t media)
2846 {
2847         dladm_status_t  status;
2848         mac_dhcpcid_t   *cid;
2849         int             i;
2850         uint_t          val_cnt = *val_cntp;
2851         val_desc_t      *vdp = *vdpp;
2852
2853         if (val_cnt > MPT_MAXCID)
2854                 return (DLADM_STATUS_BADVALCNT);
2855
2856         for (i = 0; i < val_cnt; i++) {
2857                 if ((cid = calloc(1, sizeof (mac_dhcpcid_t))) == NULL) {
2858                         status = DLADM_STATUS_NOMEM;
2859                         goto fail;
2860                 }
2861                 vdp[i].vd_val = (uintptr_t)cid;
2862
2863                 status = dladm_str2cid(prop_val[i], cid);
2864                 if (status != DLADM_STATUS_OK)
2865                         goto fail;
2866         }
2867         return (DLADM_STATUS_OK);
2868
2869 fail:
2870         for (i = 0; i < val_cnt; i++) {
2871                 free((void *)vdp[i].vd_val);
2872                 vdp[i].vd_val = NULL;
2873         }
2874         return (status);
2875 }
2876
2877 /* ARGSUSED */
2878 static dladm_status_t
2879 get_autopush(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
2880     char **prop_val, uint_t *val_cnt, datalink_media_t media,
2881     uint_t flags, uint_t *perm_flags)
2882 {
2883         struct          dlautopush dlap;
2884         int             i, len;
2885         dladm_status_t  status;
```



```
2887         if (flags & DLD_PROP_DEFAULT)
2888                 return (DLADM_STATUS_NOTDEFINED);
2889
2890         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
2891             perm_flags, &dlap, sizeof (dlap));
2892         if (status != DLADM_STATUS_OK)
2893                 return (status);
2894
2895         if (dlap.dap_npush == 0) {
2896                 *val_cnt = 0;
2897                 return (DLADM_STATUS_OK);
2898         }
2899         for (i = 0, len = 0; i < dlap.dap_npush; i++) {
2900                 if (i != 0) {
2901                         (void) snprintf(*prop_val + len,
2902                             DLADM_PROP_VAL_MAX - len, "%c", AP_DELIMITER);
2903                         len += 1;
2904                 }
2905                 (void) snprintf(*prop_val + len, DLADM_PROP_VAL_MAX - len,
2906                     "%s", dlap.dap_aplist[i]);
2907                 len += strlen(dlap.dap_aplist[i]);
2908                 if (dlap.dap_anchor - 1 == i) {
2909                         (void) snprintf(*prop_val + len,
2910                             DLADM_PROP_VAL_MAX - len, "%c%s", AP_DELIMITER,
2911                             AP_ANCHOR);
2912                         len += (strlen(AP_ANCHOR) + 1);
2913                 }
2914         }
2915         *val_cnt = 1;
2916         return (DLADM_STATUS_OK);
2917 }
2918
2919 /*
2920  * Add the specified module to the dlautopush structure; returns a
2921  * DLADM_STATUS_* code.
2922  */
2923 dladm_status_t
2924 i_dladm_add_ap_module(const char *module, struct dlautopush *dlap)
2925 {
2926         if ((strlen(module) == 0) || (strlen(module) > FMNAMESZ))
2927                 return (DLADM_STATUS_BADVAL);
2928
2929         if (strncasecmp(module, AP_ANCHOR, strlen(AP_ANCHOR)) == 0) {
2930                 /*
2931                  * We don't allow multiple anchors, and the anchor must
2932                  * be after at least one module.
2933                  */
2934                 if (dlap->dap_anchor != 0)
2935                         return (DLADM_STATUS_BADVAL);
2936                 if (dlap->dap_npush == 0)
2937                         return (DLADM_STATUS_BADVAL);
2938
2939                 dlap->dap_anchor = dlap->dap_npush;
2940                 return (DLADM_STATUS_OK);
2941         }
2942         if (dlap->dap_npush >= MAXAPUSH)
2943                 return (DLADM_STATUS_BADVALCNT);
2944
2945         (void) strlcpy(dlap->dap_aplist[dlap->dap_npush++], module,
2946             FMNAMESZ + 1);
2947
2948         return (DLADM_STATUS_OK);
2949 }
2950
2951 /*
```

```
2952  * Currently, both '.' and ' '(space) can be used as the delimiters between
2953  * autopush modules. The former is used in dladm set-linkprop, and the
2954  * latter is used in the autopush(1M) file.
2955  */
2956 /* ARGSUSED */
2957 static dladm_status_t
2958 check_autopush(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
2959     char **prop_val, uint_t *val_cntp, uint_t flags, val_desc_t **vdpp,
2960     datalink_media_t media)
2961 {
2962         char                    *module;
2963         struct dlautopush       *dlap;
2964         dladm_status_t          status;
2965         char                    val[DLADM_PROP_VAL_MAX];
2966         char                    delimiters[4];
2967         uint_t                  val_cnt = *val_cntp;
2968         val_desc_t              *vdp = *vdpp;

2970         if (val_cnt != 1)
2971                 return (DLADM_STATUS_BADVALCNT);

2973         if (prop_val != NULL) {
2974                 dlap = malloc(sizeof (struct dlautopush));
2975                 if (dlap == NULL)
2976                         return (DLADM_STATUS_NOMEM);

2978                 (void) memset(dlap, 0, sizeof (struct dlautopush));
2979                 (void) snprintf(delimiters, 4, " %c\n", AP_DELIMITER);
2980                 bcopy(*prop_val, val, DLADM_PROP_VAL_MAX);
2981                 module = strtok(val, delimiters);
2982                 while (module != NULL) {
2983                         status = i_dladm_add_ap_module(module, dlap);
2984                         if (status != DLADM_STATUS_OK)
2985                                 return (status);
2986                         module = strtok(NULL, delimiters);
2987                 }

2989                 vdp->vd_val = (uintptr_t)dlap;
2990         } else {
2991                 vdp->vd_val = 0;
2992         }
2993         return (DLADM_STATUS_OK);
2994 }

2996 #define WLDP_BUFSIZE (MAX_BUF_LEN - WIFI_BUF_OFFSET)

2998 /* ARGSUSED */
2999 static dladm_status_t
3000 get_rate_common(dladm_handle_t handle, prop_desc_t *pdp,
3001     datalink_id_t linkid, char **prop_val, uint_t *val_cnt, uint_t id,
3002     uint_t *perm_flags)
3003 {
3004         wl_rates_t      *wrp;
3005         uint_t          i;
3006         dladm_status_t  status = DLADM_STATUS_OK;

3008         wrp = malloc(WLDP_BUFSIZE);
3009         if (wrp == NULL)
3010                 return (DLADM_STATUS_NOMEM);

3012         status = i_dladm_wlan_param(handle, linkid, wrp, id, WLDP_BUFSIZE,
3013             B_FALSE);
3014         if (status != DLADM_STATUS_OK)
3015                 goto done;

3017         if (wrp->wl_rates_num > *val_cnt) {
```



```
3018                 status = DLADM_STATUS_TOOSMALL;
3019                 goto done;
3020         }

3022         if (wrp->wl_rates_rates[0] == 0) {
3023                 prop_val[0][0] = '\0';
3024                 *val_cnt = 1;
3025                 goto done;
3026         }

3028         for (i = 0; i < wrp->wl_rates_num; i++) {
3029                 (void) snprintf(prop_val[i], DLADM_STRSIZE, "%.*f",
3030                     wrp->wl_rates_rates[i] % 2,
3031                     (float)wrp->wl_rates_rates[i] / 2);
3032         }
3033         *val_cnt = wrp->wl_rates_num;
3034         *perm_flags = MAC_PROP_PERM_RW;

3036 done:
3037         free(wrp);
3038         return (status);
3039 }

3041 static dladm_status_t
3042 get_rate(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3043     char **prop_val, uint_t *val_cnt, datalink_media_t media,
3044     uint_t flags, uint_t *perm_flags)
3045 {
3046         if (media != DL_WIFI) {
3047                 return (get_speed(handle, pdp, linkid, prop_val,
3048                     val_cnt, media, flags, perm_flags));
3049         }

3051         return (get_rate_common(handle, pdp, linkid, prop_val, val_cnt,
3052             MAC_PROP_WL_DESIRED_RATES, perm_flags));
3053 }

3055 /* ARGSUSED */
3056 static dladm_status_t
3057 get_rate_mod(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3058     char **prop_val, uint_t *val_cnt, datalink_media_t media,
3059     uint_t flags, uint_t *perm_flags)
3060 {
3061         switch (media) {
3062         case DL_ETHER:
3063                 /*
3064                  * Speed for ethernet links is unbounded. E.g., 802.11b
3065                  * links can have a speed of 5.5 Gbps.
3066                  */
3067                 return (DLADM_STATUS_NOTSUP);

3069         case DL_WIFI:
3070                 return (get_rate_common(handle, pdp, linkid, prop_val,
3071                     val_cnt, MAC_PROP_WL_SUPPORTED_RATES, perm_flags));
3072         default:
3073                 return (DLADM_STATUS_BADARG);
3074         }
3075 }

3077 static dladm_status_t
3078 set_wlan_rate(dladm_handle_t handle, datalink_id_t linkid,
3079     dladm_wlan_rates_t *rates)
3080 {
3081         int             i;
3082         uint_t          len;
3083         wl_rates_t      *wrp;
```

```
3084         dladm_status_t  status = DLADM_STATUS_OK;

3086         wrp = malloc(WLDP_BUFSIZE);
3087         if (wrp == NULL)
3088                 return (DLADM_STATUS_NOMEM);

3090         bzero(wrp, WLDP_BUFSIZE);
3091         for (i = 0; i < rates->wr_cnt; i++)
3092                 wrp->wl_rates_rates[i] = rates->wr_rates[i];
3093         wrp->wl_rates_num = rates->wr_cnt;

3095         len = offsetof(wl_rates_t, wl_rates_rates) +
3096             (rates->wr_cnt * sizeof (char)) + WIFI_BUF_OFFSET;
3097         status = i_dladm_wlan_param(handle, linkid, wrp,
3098             MAC_PROP_WL_DESIRED_RATES, len, B_TRUE);

3100         free(wrp);
3101         return (status);
3102 }

3104 /* ARGSUSED */
3105 static dladm_status_t
3106 set_rate(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3107     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
3108 {
3109         dladm_wlan_rates_t      rates;
3110         dladm_status_t          status;

3112         /*
3113          * can currently set rate on WIFI links only.
3114          */
3115         if (media != DL_WIFI)
3116                 return (DLADM_STATUS_PROPRDONLY);

3118         if (val_cnt != 1)
3119                 return (DLADM_STATUS_BADVALCNT);

3121         rates.wr_cnt = 1;
3122         rates.wr_rates[0] = vdp[0].vd_val;

3124         status = set_wlan_rate(handle, linkid, &rates);

3126         return (status);
3127 }

3129 /* ARGSUSED */
3130 static dladm_status_t
3131 check_rate(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3132     char **prop_val, uint_t *val_cntp, uint_t flags, val_desc_t **vdpp,
3133     datalink_media_t media)
3134 {
3135         int             i;
3136         uint_t          modval_cnt = MAX_SUPPORT_RATES;
3137         char            *buf, **modval;
3138         dladm_status_t  status;
3139         uint_t          perm_flags;
3140         uint_t          val_cnt = *val_cntp;
3141         val_desc_t      *vdp = *vdpp;

3143         if (val_cnt != 1)
3144                 return (DLADM_STATUS_BADVALCNT);

3146         buf = malloc((sizeof (char *) + DLADM_STRSIZE) *
3147             MAX_SUPPORT_RATES);
3148         if (buf == NULL) {
3149                 status = DLADM_STATUS_NOMEM;
```



```
3150                 goto done;
3151         }

3153         modval = (char **)(void *)buf;
3154         for (i = 0; i < MAX_SUPPORT_RATES; i++) {
3155                 modval[i] = buf + sizeof (char *) * MAX_SUPPORT_RATES +
3156                     i * DLADM_STRSIZE;
3157         }

3159         status = get_rate_mod(handle, NULL, linkid, modval, &modval_cnt,
3160             media, 0, &perm_flags);
3161         if (status != DLADM_STATUS_OK)
3162                 goto done;

3164         for (i = 0; i < modval_cnt; i++) {
3165                 if (strcasecmp(*prop_val, modval[i]) == 0) {
3166                         vdp->vd_val = (uintptr_t)(uint_t)
3167                             (atof(*prop_val) * 2);
3168                         status = DLADM_STATUS_OK;
3169                         break;
3170                 }
3171         }
3172         if (i == modval_cnt)
3173                 status = DLADM_STATUS_BADVAL;
3174 done:
3175         free(buf);
3176         return (status);
3177 }

3179 static dladm_status_t
3180 get_phyconf(dladm_handle_t handle, datalink_id_t linkid, void *buf,
3181     int buflen)
3182 {
3183         return (i_dladm_wlan_param(handle, linkid, buf, MAC_PROP_WL_PHY_CONFIG,
3184             buflen, B_FALSE));
3185 }

3187 /* ARGSUSED */
3188 static dladm_status_t
3189 get_channel(dladm_handle_t handle, prop_desc_t *pdp,
3190     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3191     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3192 {
3193         uint32_t        channel;
3194         char            buf[WLDP_BUFSIZE];
3195         dladm_status_t  status;
3196         wl_phy_conf_t   wl_phy_conf;

3198         if ((status = get_phyconf(handle, linkid, buf, sizeof (buf)))
3199             != DLADM_STATUS_OK)
3200                 return (status);

3202         (void) memcpy(&wl_phy_conf, buf, sizeof (wl_phy_conf));
3203         if (!i_dladm_wlan_convert_chan(&wl_phy_conf, &channel))
3204                 return (DLADM_STATUS_NOTFOUND);

3206         (void) snprintf(*prop_val, DLADM_STRSIZE, "%u", channel);
3207         *val_cnt = 1;
3208         *perm_flags = MAC_PROP_PERM_READ;
3209         return (DLADM_STATUS_OK);
3210 }

3212 /* ARGSUSED */
3213 static dladm_status_t
3214 get_powermode(dladm_handle_t handle, prop_desc_t *pdp,
3215     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
```

```
3216     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3217 {
3218         wl_ps_mode_t    mode;
3219         const char      *s;
3220         char            buf[WLDP_BUFSIZE];
3221         dladm_status_t  status;

3223         if ((status = i_dladm_wlan_param(handle, linkid, buf,
3224             MAC_PROP_WL_POWER_MODE, sizeof (buf), B_FALSE)) != DLADM_STATUS_OK)
3225                 return (status);

3227         (void) memcpy(&mode, buf, sizeof (mode));
3228         switch (mode.wl_ps_mode) {
3229         case WL_PM_AM:
3230                 s = "off";
3231                 break;
3232         case WL_PM_MPS:
3233                 s = "max";
3234                 break;
3235         case WL_PM_FAST:
3236                 s = "fast";
3237                 break;
3238         default:
3239                 return (DLADM_STATUS_NOTFOUND);
3240         }
3241         (void) snprintf(*prop_val, DLADM_STRSIZE, "%s", s);
3242         *val_cnt = 1;
3243         *perm_flags = MAC_PROP_PERM_RW;
3244         return (DLADM_STATUS_OK);
3245 }

3247 /* ARGSUSED */
3248 static dladm_status_t
3249 set_powermode(dladm_handle_t handle, prop_desc_t *pdp,
3250     datalink_id_t linkid, val_desc_t *vdp, uint_t val_cnt, uint_t flags,
3251     datalink_media_t media)
3252 {
3253         dladm_wlan_powermode_t  powermode = vdp->vd_val;
3254         wl_ps_mode_t            ps_mode;

3256         if (val_cnt != 1)
3257                 return (DLADM_STATUS_BADVALCNT);

3259         (void) memset(&ps_mode, 0xff, sizeof (ps_mode));

3261         switch (powermode) {
3262         case DLADM_WLAN_PM_OFF:
3263                 ps_mode.wl_ps_mode = WL_PM_AM;
3264                 break;
3265         case DLADM_WLAN_PM_MAX:
3266                 ps_mode.wl_ps_mode = WL_PM_MPS;
3267                 break;
3268         case DLADM_WLAN_PM_FAST:
3269                 ps_mode.wl_ps_mode = WL_PM_FAST;
3270                 break;
3271         default:
3272                 return (DLADM_STATUS_NOTSUP);
3273         }
3274         return (i_dladm_wlan_param(handle, linkid, &ps_mode,
3275             MAC_PROP_WL_POWER_MODE, sizeof (ps_mode), B_TRUE));
3276 }

3278 /* ARGSUSED */
3279 static dladm_status_t
3280 get_radio(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3281     char **prop_val, uint_t *val_cnt, datalink_media_t media,
```



```
3282     uint_t flags, uint_t *perm_flags)
3283 {
3284         wl_radio_t      radio;
3285         const char      *s;
3286         char            buf[WLDP_BUFSIZE];
3287         dladm_status_t  status;

3289         if ((status = i_dladm_wlan_param(handle, linkid, buf,
3290             MAC_PROP_WL_RADIO, sizeof (buf), B_FALSE)) != DLADM_STATUS_OK)
3291                 return (status);

3293         (void) memcpy(&radio, buf, sizeof (radio));
3294         switch (radio) {
3295         case B_TRUE:
3296                 s = "on";
3297                 break;
3298         case B_FALSE:
3299                 s = "off";
3300                 break;
3301         default:
3302                 return (DLADM_STATUS_NOTFOUND);
3303         }
3304         (void) snprintf(*prop_val, DLADM_STRSIZE, "%s", s);
3305         *val_cnt = 1;
3306         *perm_flags = MAC_PROP_PERM_RW;
3307         return (DLADM_STATUS_OK);
3308 }

3310 /* ARGSUSED */
3311 static dladm_status_t
3312 set_radio(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3313     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
3314 {
3315         dladm_wlan_radio_t      radio = vdp->vd_val;
3316         wl_radio_t              r;

3318         if (val_cnt != 1)
3319                 return (DLADM_STATUS_BADVALCNT);

3321         switch (radio) {
3322         case DLADM_WLAN_RADIO_ON:
3323                 r = B_TRUE;
3324                 break;
3325         case DLADM_WLAN_RADIO_OFF:
3326                 r = B_FALSE;
3327                 break;
3328         default:
3329                 return (DLADM_STATUS_NOTSUP);
3330         }
3331         return (i_dladm_wlan_param(handle, linkid, &r, MAC_PROP_WL_RADIO,
3332             sizeof (r), B_TRUE));
3333 }

3335 /* ARGSUSED */
3336 static dladm_status_t
3337 check_hoplimit(dladm_handle_t handle, prop_desc_t *pdp,
3338     datalink_id_t linkid, char **prop_val, uint_t *val_cntp, uint_t flags,
3339     val_desc_t **vdpp, datalink_media_t media)
3340 {
3341         int32_t         hlim;
3342         char            *ep;
3343         uint_t          val_cnt = *val_cntp;
3344         val_desc_t      *vdp = *vdpp;

3346         if (val_cnt != 1)
3347                 return (DLADM_STATUS_BADVALCNT);
```

```
3349         errno = 0;
3350         hlim = strtol(*prop_val, &ep, 10);
3351         if (errno != 0 || ep == *prop_val || hlim < 1 ||
3352             hlim > (int32_t)UINT8_MAX)
3353                 return (DLADM_STATUS_BADVAL);
3354         vdp->vd_val = hlim;
3355         return (DLADM_STATUS_OK);
3356 }

3358 /* ARGSUSED */
3359 static dladm_status_t
3360 check_encaplim(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3361     char **prop_val, uint_t *val_cntp, uint_t flags, val_desc_t **vdpp,
3362     datalink_media_t media)
3363 {
3364         int32_t         elim;
3365         char            *ep;
3366         uint_t          val_cnt = *val_cntp;
3367         val_desc_t      *vdp = *vdpp;

3369         if (media != DL_IPV6)
3370                 return (DLADM_STATUS_BADARG);

3372         if (val_cnt != 1)
3373                 return (DLADM_STATUS_BADVALCNT);

3375         errno = 0;
3376         elim = strtol(*prop_val, &ep, 10);
3377         if (errno != 0 || ep == *prop_val || elim < 0 ||
3378             elim > (int32_t)UINT8_MAX)
3379                 return (DLADM_STATUS_BADVAL);
3380         vdp->vd_val = elim;
3381         return (DLADM_STATUS_OK);
3382 }

3384 static dladm_status_t
3385 i_dladm_set_linkprop_db(dladm_handle_t handle, datalink_id_t linkid,
3386     const char *prop_name, char **prop_val, uint_t val_cnt)
3387 {
3388         char            buf[MAXLINELEN];
3389         int             i;
3390         dladm_conf_t    conf;
3391         dladm_status_t  status;

3393         status = dladm_open_conf(handle, linkid, &conf);
3394         if (status != DLADM_STATUS_OK)
3395                 return (status);

3397         /*
3398          * reset case.
3399          */
3400         if (val_cnt == 0) {
3401                 status = dladm_unset_conf_field(handle, conf, prop_name);
3402                 if (status == DLADM_STATUS_OK)
3403                         status = dladm_write_conf(handle, conf);
3404                 goto done;
3405         }

3407         buf[0] = '\0';
3408         for (i = 0; i < val_cnt; i++) {
3409                 (void) strlcat(buf, prop_val[i], MAXLINELEN);
3410                 if (i != val_cnt - 1)
3411                         (void) strlcat(buf, ",", MAXLINELEN);
3412         }
```



```
3414         status = dladm_set_conf_field(handle, conf, prop_name, DLADM_TYPE_STR,
3415             buf);
3416         if (status == DLADM_STATUS_OK)
3417                 status = dladm_write_conf(handle, conf);

3419 done:
3420         dladm_destroy_conf(handle, conf);
3421         return (status);
3422 }

3424 static dladm_status_t
3425 i_dladm_get_linkprop_db(dladm_handle_t handle, datalink_id_t linkid,
3426     const char *prop_name, char **prop_val, uint_t *val_cntp)
3427 {
3428         char            buf[MAXLINELEN], *str;
3429         uint_t          cnt = 0;
3430         dladm_conf_t    conf;
3431         dladm_status_t  status;

3433         status = dladm_getsnap_conf(handle, linkid, &conf);
3434         if (status != DLADM_STATUS_OK)
3435                 return (status);

3437         status = dladm_get_conf_field(handle, conf, prop_name, buf, MAXLINELEN);
3438         if (status != DLADM_STATUS_OK)
3439                 goto done;

3441         str = strtok(buf, ",");
3442         while (str != NULL) {
3443                 if (cnt == *val_cntp) {
3444                         status = DLADM_STATUS_TOOSMALL;
3445                         goto done;
3446                 }
3447                 (void) strlcpy(prop_val[cnt++], str, DLADM_PROP_VAL_MAX);
3448                 str = strtok(NULL, ",");
3449         }

3451         *val_cntp = cnt;

3453 done:
3454         dladm_destroy_conf(handle, conf);
3455         return (status);
3456 }

3458 /*
3459  * Walk persistent private link properties of a link.
3460  */
3461 static dladm_status_t
3462 i_dladm_walk_linkprop_priv_db(dladm_handle_t handle, datalink_id_t linkid,
3463     void *arg, int (*func)(dladm_handle_t, datalink_id_t, const char *, void *))
3464 {
3465         dladm_status_t          status;
3466         dladm_conf_t            conf;
3467         char                    last_attr[MAXLINKATTRLEN];
3468         char                    attr[MAXLINKATTRLEN];
3469         char                    attrval[MAXLINKATTRVALLEN];
3470         size_t                  attrsz;

3472         if (linkid == DATALINK_INVALID_LINKID || func == NULL)
3473                 return (DLADM_STATUS_BADARG);

3475         status = dladm_getsnap_conf(handle, linkid, &conf);
3476         if (status != DLADM_STATUS_OK)
3477                 return (status);

3479         last_attr[0] = '\0';
```

```
3480         while ((status = dladm_getnext_conf_linkprop(handle, conf, last_attr,
3481             attr, attrval, MAXLINKATTRVALLEN, &attrsz)) == DLADM_STATUS_OK) {
3482                 if (attr[0] == '_') {
3483                         if (func(handle, linkid, attr, arg) ==
3484                             DLADM_WALK_TERMINATE)
3485                                 break;
3486                 }
3487                 (void) strlcpy(last_attr, attr, MAXLINKATTRLEN);
3488         }

3490         dladm_destroy_conf(handle, conf);
3491         return (DLADM_STATUS_OK);
3492 }

3494 static link_attr_t *
3495 dladm_name2prop(const char *prop_name)
3496 {
3497         link_attr_t *p;

3499         for (p = link_attr; p->pp_id != MAC_PROP_PRIVATE; p++) {
3500                 if (strcmp(p->pp_name, prop_name) == 0)
3501                         break;
3502         }
3503         return (p);
3504 }

3506 static link_attr_t *
3507 dladm_id2prop(mac_prop_id_t propid)
3508 {
3509         link_attr_t *p;

3511         for (p = link_attr; p->pp_id != MAC_PROP_PRIVATE; p++) {
3512                 if (p->pp_id == propid)
3513                         break;
3514         }
3515         return (p);
3516 }

3518 static dld_ioc_macprop_t *
3519 i_dladm_buf_alloc_impl(size_t valsize, datalink_id_t linkid,
3520     const char *prop_name, mac_prop_id_t propid, uint_t flags,
3521     dladm_status_t *status)
3522 {
3523         int dsize;
3524         dld_ioc_macprop_t *dip;

3526         *status = DLADM_STATUS_OK;
3527         dsize = MAC_PROP_BUFSIZE(valsize);
3528         dip = malloc(dsize);
3529         if (dip == NULL) {
3530                 *status = DLADM_STATUS_NOMEM;
3531                 return (NULL);
3532         }
3533         bzero(dip, dsize);
3534         dip->pr_valsize = valsize;
3535         (void) strlcpy(dip->pr_name, prop_name, sizeof (dip->pr_name));
3536         dip->pr_linkid = linkid;
3537         dip->pr_num = propid;
3538         dip->pr_flags = flags;
3539         return (dip);
3540 }

3542 static dld_ioc_macprop_t *
3543 i_dladm_buf_alloc_by_name(size_t valsize, datalink_id_t linkid,
3544     const char *prop_name, uint_t flags, dladm_status_t *status)
3545 {
```



```
3546         link_attr_t *p;

3548         p = dladm_name2prop(prop_name);
3549         valsize = MAX(p->pp_valsize, valsize);
3550         return (i_dladm_buf_alloc_impl(valsize, linkid, prop_name, p->pp_id,
3551             flags, status));
3552 }

3554 static dld_ioc_macprop_t *
3555 i_dladm_buf_alloc_by_id(size_t valsize, datalink_id_t linkid,
3556     mac_prop_id_t propid, uint_t flags, dladm_status_t *status)
3557 {
3558         link_attr_t *p;

3560         p = dladm_id2prop(propid);
3561         valsize = MAX(p->pp_valsize, valsize);
3562         return (i_dladm_buf_alloc_impl(valsize, linkid, p->pp_name, propid,
3563             flags, status));
3564 }

3566 /* ARGSUSED */
3567 static dladm_status_t
3568 set_public_prop(dladm_handle_t handle, prop_desc_t *pdp,
3569     datalink_id_t linkid, val_desc_t *vdp, uint_t val_cnt, uint_t flags,
3570     datalink_media_t media)
3571 {
3572         dld_ioc_macprop_t       *dip;
3573         dladm_status_t  status = DLADM_STATUS_OK;
3574         uint8_t         u8;
3575         uint16_t        u16;
3576         uint32_t        u32;
3577         void            *val;

3579         dip = i_dladm_buf_alloc_by_name(0, linkid, pdp->pd_name, 0, &status);
3580         if (dip == NULL)
3581                 return (status);

3583         if (pdp->pd_flags & PD_CHECK_ALLOC)
3584                 val = (void *)vdp->vd_val;
3585         else {
3586                 /*
3587                  * Currently all 1/2/4-byte size properties are byte/word/int.
3588                  * No need (yet) to distinguish these from arrays of same size.
3589                  */
3590                 switch (dip->pr_valsize) {
3591                 case 1:
3592                         u8 = vdp->vd_val;
3593                         val = &u8;
3594                         break;
3595                 case 2:
3596                         u16 = vdp->vd_val;
3597                         val = &u16;
3598                         break;
3599                 case 4:
3600                         u32 = vdp->vd_val;
3601                         val = &u32;
3602                         break;
3603                 default:
3604                         val = &vdp->vd_val;
3605                         break;
3606                 }
3607         }

3609         if (val != NULL)
3610                 (void) memcpy(dip->pr_val, val, dip->pr_valsize);
3611         else
```

```
3612                 dip->pr_valsize = 0;

3614         status = i_dladm_macprop(handle, dip, B_TRUE);

3616 done:
3617         free(dip);
3618         return (status);
3619 }

3621 dladm_status_t
3622 i_dladm_macprop(dladm_handle_t handle, void *dip, boolean_t set)
3623 {
3624         dladm_status_t status = DLADM_STATUS_OK;

3626         if (ioctl(dladm_dld_fd(handle),
3627             (set ? DLDIOC_SETMACPROP : DLDIOC_GETMACPROP), dip))
3628                 status = dladm_errno2status(errno);

3630         return (status);
3631 }

3633 static dladm_status_t
3634 i_dladm_get_public_prop(dladm_handle_t handle, datalink_id_t linkid,
3635     char *prop_name, uint_t flags, uint_t *perm_flags, void *arg, size_t size)
3636 {
3637         dld_ioc_macprop_t       *dip;
3638         dladm_status_t          status;

3640         dip = i_dladm_buf_alloc_by_name(0, linkid, prop_name, flags, &status);
3641         if (dip == NULL)
3642                 return (DLADM_STATUS_NOMEM);

3644         status = i_dladm_macprop(handle, dip, B_FALSE);
3645         if (status != DLADM_STATUS_OK) {
3646                 free(dip);
3647                 return (status);
3648         }

3650         if (perm_flags != NULL)
3651                 *perm_flags = dip->pr_perm_flags;

3653         if (arg != NULL)
3654                 (void) memcpy(arg, dip->pr_val, size);
3655         free(dip);
3656         return (DLADM_STATUS_OK);
3657 }

3659 /* ARGSUSED */
3660 static dladm_status_t
3661 check_uint32(dladm_handle_t handle, prop_desc_t *pdp,
3662     datalink_id_t linkid, char **prop_val, uint_t *val_cntp, uint_t flags,
3663     val_desc_t **vp, datalink_media_t media)
3664 {
3665         uint_t          val_cnt = *val_cntp;
3666         val_desc_t      *v = *vp;

3668         if (val_cnt != 1)
3669                 return (DLADM_STATUS_BADVAL);
3670         v->vd_val = strtoul(prop_val[0], NULL, 0);
3671         return (DLADM_STATUS_OK);
3672 }

3674 /* ARGSUSED */
3675 static dladm_status_t
3676 get_duplex(dladm_handle_t handle, prop_desc_t *pdp,
3677     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
```



```
3678     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3679 {
3680         link_duplex_t   link_duplex;
3681         dladm_status_t  status;

3683         if ((status = dladm_get_single_mac_stat(handle, linkid, "link_duplex",
3684             KSTAT_DATA_UINT32, &link_duplex)) != 0)
3685                 return (status);

3687         switch (link_duplex) {
3688         case LINK_DUPLEX_FULL:
3689                 (void) strcpy(*prop_val, "full");
3690                 break;
3691         case LINK_DUPLEX_HALF:
3692                 (void) strcpy(*prop_val, "half");
3693                 break;
3694         default:
3695                 (void) strcpy(*prop_val, "unknown");
3696                 break;
3697         }
3698         *val_cnt = 1;
3699         return (DLADM_STATUS_OK);
3700 }

3702 /* ARGSUSED */
3703 static dladm_status_t
3704 get_speed(dladm_handle_t handle, prop_desc_t *pdp, datalink_id_t linkid,
3705     char **prop_val, uint_t *val_cnt, datalink_media_t media, uint_t flags,
3706     uint_t *perm_flags)
3707 {
3708         uint64_t        ifspeed = 0;
3709         dladm_status_t status;

3711         if ((status = dladm_get_single_mac_stat(handle, linkid, "ifspeed",
3712             KSTAT_DATA_UINT64, &ifspeed)) != 0)
3713                 return (status);

3715         if ((ifspeed % 1000000) != 0) {
3716                 (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX,
3717                     "%llf", ifspeed / (float)1000000); /* Mbps */
3718         } else {
3719                 (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX,
3720                     "%llu", ifspeed / 1000000); /* Mbps */
3721         }
3722         *val_cnt = 1;
3723         *perm_flags = MAC_PROP_PERM_READ;
3724         return (DLADM_STATUS_OK);
3725 }

3727 /* ARGSUSED */
3728 static dladm_status_t
3729 get_link_state(dladm_handle_t handle, prop_desc_t *pdp,
3730     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3731     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3732 {
3733         link_state_t            link_state;
3734         dladm_status_t          status;

3736         status = dladm_get_state(handle, linkid, &link_state);
3737         if (status != DLADM_STATUS_OK)
3738                 return (status);

3740         switch (link_state) {
3741         case LINK_STATE_UP:
3742                 (void) strcpy(*prop_val, "up");
3743                 break;
```

```
3744         case LINK_STATE_DOWN:
3745                 (void) strcpy(*prop_val, "down");
3746                 break;
3747         default:
3748                 (void) strcpy(*prop_val, "unknown");
3749                 break;
3750         }
3751         *val_cnt = 1;
3752         *perm_flags = MAC_PROP_PERM_READ;
3753         return (DLADM_STATUS_OK);
3754 }

3756 /* ARGSUSED */
3757 static dladm_status_t
3758 get_binary(dladm_handle_t handle, prop_desc_t *pdp,
3759     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3760     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3761 {
3762         dladm_status_t  status;
3763         uint_t          v = 0;

3765         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
3766             perm_flags, &v, sizeof (v));
3767         if (status != DLADM_STATUS_OK)
3768                 return (status);

3770         (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX, "%d", (uint_t)(v > 0));
3771         *val_cnt = 1;
3772         return (DLADM_STATUS_OK);
3773 }

3775 /* ARGSUSED */
3776 static dladm_status_t
3777 get_uint32(dladm_handle_t handle, prop_desc_t *pdp,
3778     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3779     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3780 {
3781         dladm_status_t  status;
3782         uint32_t        v = 0;

3784         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
3785             perm_flags, &v, sizeof (v));
3786         if (status != DLADM_STATUS_OK)
3787                 return (status);

3789         (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX, "%ld", v);
3790         *val_cnt = 1;
3791         return (DLADM_STATUS_OK);
3792 }

3794 /* ARGSUSED */
3795 static dladm_status_t
3796 get_range(dladm_handle_t handle, prop_desc_t *pdp,
3797     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3798     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3799 {
3800         dld_ioc_macprop_t *dip;
3801         dladm_status_t status = DLADM_STATUS_OK;
3802         size_t  sz;
3803         uint_t  rcount;
3804         mac_propval_range_t *rangep;

3806         /*
3807          * As caller we don't know number of value ranges, the driver
3808          * supports. To begin with we assume that number to be 1. If the
3809          * buffer size is insufficient, driver returns back with the
```



```
3810          * actual count of value ranges. See mac.h for more details.
3811          */
3812         sz = sizeof (mac_propval_range_t);
3813         rcount = 1;
3814 retry:
3815         if ((dip = i_dladm_buf_alloc_by_name(sz, linkid, pdp->pd_name, flags,
3816             &status)) == NULL)
3817                 return (status);

3819         rangep = (mac_propval_range_t *)(void *)&dip->pr_val;
3820         rangep->mpr_count = rcount;

3822         status = i_dladm_macprop(handle, dip, B_FALSE);
3823         if (status != DLADM_STATUS_OK) {
3824                 if (status == DLADM_STATUS_TOOSMALL) {
3825                         int err;

3827                         if ((err = i_dladm_range_size(rangep, &sz, &rcount))
3828                             == 0) {
3829                                 free(dip);
3830                                 goto retry;
3831                         } else {
3832                                 status = dladm_errno2status(err);
3833                         }
3834                 }
3835                 free(dip);
3836                 return (status);
3837         }

3839         if (rangep->mpr_count == 0) {
3840                 *val_cnt = 1;
3841                 (void) snprintf(prop_val[0], DLADM_PROP_VAL_MAX, "--");
3842                 goto done;
3843         }

3845         switch (rangep->mpr_type) {
3846         case MAC_PROPVAL_UINT32: {
3847                 mac_propval_uint32_range_t *ur;
3848                 uint_t  count = rangep->mpr_count, i;

3850                 ur = &rangep->mpr_range_uint32[0];

3852                 for (i = 0; i < count; i++, ur++) {
3853                         if (ur->mpur_min == ur->mpur_max) {
3854                                 (void) snprintf(prop_val[i], DLADM_PROP_VAL_MAX,
3855                                     "%ld", ur->mpur_min);
3856                         } else {
3857                                 (void) snprintf(prop_val[i], DLADM_PROP_VAL_MAX,
3858                                     "%ld-%ld", ur->mpur_min, ur->mpur_max);
3859                         }
3860                 }
3861                 *val_cnt = count;
3862                 break;
3863         }
3864         default:
3865                 status = DLADM_STATUS_BADARG;
3866                 break;
3867         }
3868 done:
3869         free(dip);
3870         return (status);
3871 }

3873 /* ARGSUSED */
3874 static dladm_status_t
3875 get_tagmode(dladm_handle_t handle, prop_desc_t *pdp,
```

```
3876     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3877     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3878 {
3879         link_tagmode_t          mode;
3880         dladm_status_t          status;

3882         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
3883             perm_flags, &mode, sizeof (mode));
3884         if (status != DLADM_STATUS_OK)
3885                 return (status);

3887         switch (mode) {
3888         case LINK_TAGMODE_NORMAL:
3889                 (void) strlcpy(*prop_val, "normal", DLADM_PROP_VAL_MAX);
3890                 break;
3891         case LINK_TAGMODE_VLANONLY:
3892                 (void) strlcpy(*prop_val, "vlanonly", DLADM_PROP_VAL_MAX);
3893                 break;
3894         default:
3895                 (void) strlcpy(*prop_val, "unknown", DLADM_PROP_VAL_MAX);
3896         }
3897         *val_cnt = 1;
3898         return (DLADM_STATUS_OK);
3899 }

3901 /* ARGSUSED */
3902 static dladm_status_t
3903 get_flowctl(dladm_handle_t handle, prop_desc_t *pdp,
3904     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
3905     datalink_media_t media, uint_t flags, uint_t *perm_flags)
3906 {
3907         link_flowctrl_t v;
3908         dladm_status_t  status;

3910         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
3911             perm_flags, &v, sizeof (v));
3912         if (status != DLADM_STATUS_OK)
3913                 return (status);

3915         switch (v) {
3916         case LINK_FLOWCTRL_NONE:
3917                 (void) sprintf(*prop_val, "no");
3918                 break;
3919         case LINK_FLOWCTRL_RX:
3920                 (void) sprintf(*prop_val, "rx");
3921                 break;
3922         case LINK_FLOWCTRL_TX:
3923                 (void) sprintf(*prop_val, "tx");
3924                 break;
3925         case LINK_FLOWCTRL_BI:
3926                 (void) sprintf(*prop_val, "bi");
3927                 break;
3928         }
3929         *val_cnt = 1;
3930         return (DLADM_STATUS_OK);
3931 }


3934 /* ARGSUSED */
3935 static dladm_status_t
3936 i_dladm_set_private_prop(dladm_handle_t handle, datalink_id_t linkid,
3937     const char *prop_name, char **prop_val, uint_t val_cnt, uint_t flags)

3939 {
3940         int             i, slen;
3941         int             bufsize = 0;
```



```
3942         dld_ioc_macprop_t *dip = NULL;
3943         uchar_t         *dp;
3944         link_attr_t *p;
3945         dladm_status_t  status = DLADM_STATUS_OK;

3947         if ((prop_name == NULL && prop_val != NULL) ||
3948             (prop_val != NULL && val_cnt == 0))
3949                 return (DLADM_STATUS_BADARG);
3950         p = dladm_name2prop(prop_name);
3951         if (p->pp_id != MAC_PROP_PRIVATE)
3952                 return (DLADM_STATUS_BADARG);

3954         if (!(flags & DLADM_OPT_ACTIVE))
3955                 return (DLADM_STATUS_OK);

3957         /*
3958          * private properties: all parsing is done in the kernel.
3959          * allocate a enough space for each property + its separator (',').
3960          */
3961         for (i = 0; i < val_cnt; i++) {
3962                 bufsize += strlen(prop_val[i]) + 1;
3963         }

3965         if (prop_val == NULL) {
3966                 /*
3967                  * getting default value. so use more buffer space.
3968                  */
3969                 bufsize += DLADM_PROP_BUF_CHUNK;
3970         }

3972         dip = i_dladm_buf_alloc_by_name(bufsize + 1, linkid, prop_name,
3973             (prop_val != NULL ? 0 : DLD_PROP_DEFAULT), &status);
3974         if (dip == NULL)
3975                 return (status);

3977         dp = (uchar_t *)dip->pr_val;
3978         slen = 0;

3980         if (prop_val == NULL) {
3981                 status = i_dladm_macprop(handle, dip, B_FALSE);
3982                 dip->pr_flags = 0;
3983         } else {
3984                 for (i = 0; i < val_cnt; i++) {
3985                         int plen = 0;

3987                         plen = strlen(prop_val[i]);
3988                         bcopy(prop_val[i], dp, plen);
3989                         slen += plen;
3990                         /*
3991                          * add a "," separator and update dp.
3992                          */
3993                         if (i != (val_cnt -1))
3994                                 dp[slen++] = ',';
3995                         dp += (plen + 1);
3996                 }
3997         }
3998         if (status == DLADM_STATUS_OK)
3999                 status = i_dladm_macprop(handle, dip, B_TRUE);

4001         free(dip);
4002         return (status);
4003 }

4005 static dladm_status_t
4006 i_dladm_get_priv_prop(dladm_handle_t handle, datalink_id_t linkid,
4007     const char *prop_name, char **prop_val, uint_t *val_cnt,
```

```
4008     dladm_prop_type_t type, uint_t dld_flags)
4009 {
4010         dladm_status_t  status = DLADM_STATUS_OK;
4011         dld_ioc_macprop_t *dip = NULL;
4012         link_attr_t *p;
4013
4014         if ((prop_name == NULL && prop_val != NULL) ||
4015             (prop_val != NULL && val_cnt == 0))
4016                 return (DLADM_STATUS_BADARG);
4017
4018         p = dladm_name2prop(prop_name);
4019         if (p->pp_id != MAC_PROP_PRIVATE)
4020                 return (DLADM_STATUS_BADARG);
4021
4022         /*
4023          * private properties: all parsing is done in the kernel.
4024          */
4025         dip = i_dladm_buf_alloc_by_name(DLADM_PROP_BUF_CHUNK, linkid, prop_name,
4026             dld_flags, &status);
4027         if (dip == NULL)
4028                 return (status);
4029
4030         if ((status = i_dladm_macprop(handle, dip, B_FALSE)) ==
4031             DLADM_STATUS_OK) {
4032                 if (type == DLADM_PROP_VAL_PERM) {
4033                         (void) dladm_perm2str(dip->pr_perm_flags, *prop_val);
4034                 } else if (type == DLADM_PROP_VAL_MODIFIABLE) {
4035                         *prop_val[0] = '\0';
4036                 } else {
4037                         (void) strncpy(*prop_val, dip->pr_val,
4038                             DLADM_PROP_VAL_MAX);
4039                 }
4040                 *val_cnt = 1;
4041         } else if ((status == DLADM_STATUS_NOTSUP) &&
4042             (type == DLADM_PROP_VAL_CURRENT)) {
4043                 status = DLADM_STATUS_NOTFOUND;
4044         }
4045         free(dip);
4046         return (status);
4047 }
4048
4049
4050 static dladm_status_t
4051 i_dladm_getset_defval(dladm_handle_t handle, prop_desc_t *pdp,
4052     datalink_id_t linkid, datalink_media_t media, uint_t flags)
4053 {
4054         dladm_status_t status;
4055         char **prop_vals = NULL, *buf;
4056         size_t bufsize;
4057         uint_t cnt;
4058         int i;
4059         uint_t perm_flags;
4060
4061         /*
4062          * Allocate buffer needed for prop_vals array. We can have at most
4063          * DLADM_MAX_PROP_VALCNT char *prop_vals[] entries, where
4064          * each entry has max size DLADM_PROP_VAL_MAX
4065          */
4066         bufsize =
4067             (sizeof (char *) + DLADM_PROP_VAL_MAX) * DLADM_MAX_PROP_VALCNT;
4068         buf = malloc(bufsize);
4069         prop_vals = (char **)(void *)buf;
4070         for (i = 0; i < DLADM_MAX_PROP_VALCNT; i++) {
4071                 prop_vals[i] = buf +
4072                     sizeof (char *) * DLADM_MAX_PROP_VALCNT +
4073                     i * DLADM_PROP_VAL_MAX;
```



```
4074         }
4075
4076         /*
4077          * For properties which have pdp->pd_defval.vd_name as a non-empty
4078          * string, the "" itself is used to reset the property (exceptions
4079          * are zone and autopush, which populate vdp->vd_val). So
4080          * libdladm can copy pdp->pd_defval over to the val_desc_t passed
4081          * down on the setprop using the global values in the table. For
4082          * other cases (vd_name is ""), doing reset-linkprop will cause
4083          * libdladm to do a getprop to find the default value and then do
4084          * a setprop to reset the value to default.
4085          */
4086         status = pdp->pd_get(handle, pdp, linkid, prop_vals, &cnt, media,
4087             DLD_PROP_DEFAULT, &perm_flags);
4088         if (status == DLADM_STATUS_OK) {
4089                 if (perm_flags == MAC_PROP_PERM_RW) {
4090                         status = i_dladm_set_single_prop(handle, linkid,
4091                             pdp->pd_class, media, pdp, prop_vals, cnt, flags);
4092                 }
4093                 else
4094                         status = DLADM_STATUS_NOTSUP;
4095         }
4096         free(buf);
4097         return (status);
4098 }
4099
4100 /* ARGSUSED */
4101 static dladm_status_t
4102 get_stp(dladm_handle_t handle, struct prop_desc *pd, datalink_id_t linkid,
4103     char **prop_val, uint_t *val_cnt, datalink_media_t media, uint_t flags,
4104     uint_t *perm_flags)
4105 {
4106         const bridge_public_prop_t *bpp;
4107         dladm_status_t retv;
4108         int val, i;
4109
4110         if (flags != 0)
4111                 return (DLADM_STATUS_NOTSUP);
4112         *perm_flags = MAC_PROP_PERM_RW;
4113         *val_cnt = 1;
4114         for (bpp = bridge_prop; bpp->bpp_name != NULL; bpp++)
4115                 if (strcmp(bpp->bpp_name, pd->pd_name) == 0)
4116                         break;
4117         retv = dladm_bridge_get_port_cfg(handle, linkid, bpp->bpp_code, &val);
4118         /* If the daemon isn't running, then return the persistent value */
4119         if (retv == DLADM_STATUS_NOTFOUND) {
4120                 if (i_dladm_get_linkprop_db(handle, linkid, pd->pd_name,
4121                     prop_val, val_cnt) != DLADM_STATUS_OK)
4122                         (void) strlcpy(*prop_val, pd->pd_defval.vd_name,
4123                             DLADM_PROP_VAL_MAX);
4124                 return (DLADM_STATUS_OK);
4125         }
4126         if (retv != DLADM_STATUS_OK) {
4127                 (void) strlcpy(*prop_val, "?", DLADM_PROP_VAL_MAX);
4128                 return (retv);
4129         }
4130         if (val == pd->pd_defval.vd_val && pd->pd_defval.vd_name[0] != '\0') {
4131                 (void) strlcpy(*prop_val, pd->pd_defval.vd_name,
4132                     DLADM_PROP_VAL_MAX);
4133                 return (DLADM_STATUS_OK);
4134         }
4135         for (i = 0; i < pd->pd_noptval; i++) {
4136                 if (val == pd->pd_optval[i].vd_val) {
4137                         (void) strlcpy(*prop_val, pd->pd_optval[i].vd_name,
4138                             DLADM_PROP_VAL_MAX);
4139                         return (DLADM_STATUS_OK);
```

```
4140                 }
4141         }
4142         (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX, "%u", (unsigned)val);
4143         return (DLADM_STATUS_OK);
4144 }

4146 /* ARGSUSED1 */
4147 static dladm_status_t
4148 set_stp_prop(dladm_handle_t handle, prop_desc_t *pd, datalink_id_t linkid,
4149     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
4150 {
4151         /*
4152          * Special case for mcheck: the daemon resets the value to zero, and we
4153          * don't want the daemon to refresh itself; it leads to deadlock.
4154          */
4155         if (flags & DLADM_OPT_NOREFRESH)
4156                 return (DLADM_STATUS_OK);

4158         /* Tell the running daemon, if any */
4159         return (dladm_bridge_refresh(handle, linkid));
4160 }

4162 /*
4163  * This is used only for stp_priority, stp_cost, and stp_mcheck.
4164  */
4165 /* ARGSUSED */
4166 static dladm_status_t
4167 check_stp_prop(dladm_handle_t handle, struct prop_desc *pd,
4168     datalink_id_t linkid, char **prop_val, uint_t *val_cntp, uint_t flags,
4169     val_desc_t **vdpp, datalink_media_t media)
4170 {
4171         char            *cp;
4172         boolean_t       iscost;
4173         uint_t          val_cnt = *val_cntp;
4174         val_desc_t      *vdp = *vdpp;

4176         if (val_cnt != 1)
4177                 return (DLADM_STATUS_BADVALCNT);

4179         if (prop_val == NULL) {
4180                 vdp->vd_val = 0;
4181         } else {
4182                 /* Only stp_priority and stp_cost use this function */
4183                 iscost = strcmp(pd->pd_name, "stp_cost") == 0;

4185                 if (iscost && strcmp(prop_val[0], "auto") == 0) {
4186                         /* Illegal value 0 is allowed to mean "automatic" */
4187                         vdp->vd_val = 0;
4188                 } else {
4189                         errno = 0;
4190                         vdp->vd_val = strtoul(prop_val[0], &cp, 0);
4191                         if (errno != 0 || *cp != '\0')
4192                                 return (DLADM_STATUS_BADVAL);
4193                 }
4194         }

4196         if (iscost) {
4197                 return (vdp->vd_val > 65535 ? DLADM_STATUS_BADVAL :
4198                     DLADM_STATUS_OK);
4199         } else {
4200                 if (vdp->vd_val > 255)
4201                         return (DLADM_STATUS_BADVAL);
4202                 /*
4203                  * If the user is setting stp_mcheck non-zero, then (per the
4204                  * IEEE management standards and UNH testing) we need to check
4205                  * whether this link is part of a bridge that is running RSTP.
```



```
4206                  * If it's not, then setting the flag is an error.  Note that
4207                  * errors are intentionally discarded here; it's the value
4208                  * that's the problem -- it's not a bad value, merely one that
4209                  * can't be used now.
4210                  */
4211                 if (strcmp(pd->pd_name, "stp_mcheck") == 0 &&
4212                     vdp->vd_val != 0) {
4213                         char bridge[MAXLINKNAMELEN];
4214                         UID_STP_CFG_T cfg;
4215                         dladm_bridge_prot_t brprot;

4217                         if (dladm_bridge_getlink(handle, linkid, bridge,
4218                             sizeof (bridge)) != DLADM_STATUS_OK ||
4219                             dladm_bridge_get_properties(bridge, &cfg,
4220                             &brprot) != DLADM_STATUS_OK)
4221                                 return (DLADM_STATUS_FAILED);
4222                         if (cfg.force_version <= 1)
4223                                 return (DLADM_STATUS_FAILED);
4224                 }
4225                 return (DLADM_STATUS_OK);
4226         }
4227 }

4229 /* ARGSUSED */
4230 static dladm_status_t
4231 get_bridge_forward(dladm_handle_t handle, struct prop_desc *pd,
4232     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
4233     datalink_media_t media, uint_t flags, uint_t *perm_flags)
4234 {
4235         dladm_status_t retv;
4236         uint_t val;

4238         if (flags != 0)
4239                 return (DLADM_STATUS_NOTSUP);
4240         *perm_flags = MAC_PROP_PERM_RW;
4241         *val_cnt = 1;
4242         retv = dladm_bridge_get_forwarding(handle, linkid, &val);
4243         if (retv == DLADM_STATUS_NOTFOUND) {
4244                 if (i_dladm_get_linkprop_db(handle, linkid, pd->pd_name,
4245                     prop_val, val_cnt) != DLADM_STATUS_OK)
4246                         (void) strlcpy(*prop_val, pd->pd_defval.vd_name,
4247                             DLADM_PROP_VAL_MAX);
4248                 return (DLADM_STATUS_OK);
4249         }
4250         if (retv == DLADM_STATUS_OK)
4251                 (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX, "%u", val);
4252         else
4253                 (void) strlcpy(*prop_val, "?", DLADM_PROP_VAL_MAX);
4254         return (retv);
4255 }

4257 /* ARGSUSED */
4258 static dladm_status_t
4259 set_bridge_forward(dladm_handle_t handle, prop_desc_t *pd, datalink_id_t linkid,
4260     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
4261 {
4262         /* Tell the running daemon, if any */
4263         return (dladm_bridge_refresh(handle, linkid));
4264 }

4266 /* ARGSUSED */
4267 static dladm_status_t
4268 get_bridge_pvid(dladm_handle_t handle, struct prop_desc *pd,
4269     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
4270     datalink_media_t media, uint_t flags, uint_t *perm_flags)
4271 {
```

```
4272         dladm_status_t status;
4273         dld_ioc_macprop_t *dip;
4274         uint16_t pvid;

4276         if (flags != 0)
4277                 return (DLADM_STATUS_NOTSUP);
4278         *perm_flags = MAC_PROP_PERM_RW;
4279         *val_cnt = 1;
4280         dip = i_dladm_buf_alloc_by_id(sizeof (uint16_t), linkid, MAC_PROP_PVID,
4281             0, &status);
4282         if (dip == NULL)
4283                 return (status);
4284         status = i_dladm_macprop(handle, dip, B_FALSE);
4285         if (status == DLADM_STATUS_OK) {
4286                 (void) memcpy(&pvid, dip->pr_val, sizeof (pvid));
4287                 (void) snprintf(*prop_val, DLADM_PROP_VAL_MAX, "%u", pvid);
4288         } else {
4289                 (void) strlcpy(*prop_val, "?", DLADM_PROP_VAL_MAX);
4290         }
4291         free(dip);
4292         return (status);
4293 }

4295 /* ARGSUSED */
4296 static dladm_status_t
4297 set_bridge_pvid(dladm_handle_t handle, prop_desc_t *pd, datalink_id_t linkid,
4298     val_desc_t *vdp, uint_t val_cnt, uint_t flags, datalink_media_t media)
4299 {
4300         dladm_status_t status;
4301         dld_ioc_macprop_t *dip;
4302         uint16_t pvid;

4304         dip = i_dladm_buf_alloc_by_id(sizeof (uint16_t), linkid, MAC_PROP_PVID,
4305             0, &status);
4306         if (dip == NULL)
4307                 return (status);
4308         pvid = vdp->vd_val;
4309         (void) memcpy(dip->pr_val, &pvid, sizeof (pvid));
4310         status = i_dladm_macprop(handle, dip, B_TRUE);
4311         free(dip);
4312         if (status != DLADM_STATUS_OK)
4313                 return (status);

4315         /* Tell the running daemon, if any */
4316         return (dladm_bridge_refresh(handle, linkid));
4317 }

4319 /* ARGSUSED */
4320 static dladm_status_t
4321 check_bridge_pvid(dladm_handle_t handle, struct prop_desc *pd,
4322     datalink_id_t linkid, char **prop_val, uint_t *val_cntp, uint_t flags,
4323     val_desc_t **vdpp, datalink_media_t media)
4324 {
4325         char            *cp;
4326         uint_t          val_cnt = *val_cntp;
4327         val_desc_t      *vdp = *vdpp;

4329         if (val_cnt != 1)
4330                 return (DLADM_STATUS_BADVALCNT);

4332         if (prop_val == NULL) {
4333                 vdp->vd_val = 1;
4334         } else {
4335                 errno = 0;
4336                 vdp->vd_val = strtoul(prop_val[0], &cp, 0);
4337                 if (errno != 0 || *cp != '\0')
```



```
4338                         return (DLADM_STATUS_BADVAL);
4339         }

4341         return (vdp->vd_val > VLAN_ID_MAX ? DLADM_STATUS_BADVAL :
4342             DLADM_STATUS_OK);
4343 }

4345 dladm_status_t
4346 i_dladm_wlan_param(dladm_handle_t handle, datalink_id_t linkid, void *buf,
4347     mac_prop_id_t cmd, size_t len, boolean_t set)
4348 {
4349         uint32_t                flags;
4350         dladm_status_t          status;
4351         uint32_t                media;
4352         dld_ioc_macprop_t       *dip;
4353         void                    *dp;

4355         if ((status = dladm_datalink_id2info(handle, linkid, &flags, NULL,
4356             &media, NULL, 0)) != DLADM_STATUS_OK) {
4357                 return (status);
4358         }

4360         if (media != DL_WIFI)
4361                 return (DLADM_STATUS_BADARG);

4363         if (!(flags & DLADM_OPT_ACTIVE))
4364                 return (DLADM_STATUS_TEMPONLY);

4366         if (len == (MAX_BUF_LEN - WIFI_BUF_OFFSET))
4367                 len = MAX_BUF_LEN - sizeof (dld_ioc_macprop_t) - 1;

4369         dip = i_dladm_buf_alloc_by_id(len, linkid, cmd, 0, &status);
4370         if (dip == NULL)
4371                 return (DLADM_STATUS_NOMEM);

4373         dp = (uchar_t *)dip->pr_val;
4374         if (set)
4375                 (void) memcpy(dp, buf, len);

4377         status = i_dladm_macprop(handle, dip, set);
4378         if (status == DLADM_STATUS_OK) {
4379                 if (!set)
4380                         (void) memcpy(buf, dp, len);
4381         }

4383         free(dip);
4384         return (status);
4385 }

4387 dladm_status_t
4388 dladm_parse_link_props(char *str, dladm_arg_list_t **listp, boolean_t novalues)
4389 {
4390         return (dladm_parse_args(str, listp, novalues));
4391 }

4393 /*
4394  * Retrieve the one link property from the database
4395  */
4396 /*ARGSUSED*/
4397 static int
4398 i_dladm_get_one_prop(dladm_handle_t handle, datalink_id_t linkid,
4399     const char *prop_name, void *arg)
4400 {
4401         dladm_arg_list_t        *proplist = arg;
4402         dladm_arg_info_t        *aip = NULL;
```

```
4404         aip = &proplist->al_info[proplist->al_count];
4405         /*
4406          * it is fine to point to prop_name since prop_name points to the
4407          * prop_table[n].pd_name.
4408          */
4409         aip->ai_name = prop_name;

4411         (void) dladm_get_linkprop(handle, linkid, DLADM_PROP_VAL_PERSISTENT,
4412             prop_name, aip->ai_val, &aip->ai_count);

4414         if (aip->ai_count != 0)
4415                 proplist->al_count++;

4417         return (DLADM_WALK_CONTINUE);
4418 }


4421 /*
4422  * Retrieve all link properties for a link from the database and
4423  * return a property list.
4424  */
4425 dladm_status_t
4426 dladm_link_get_proplist(dladm_handle_t handle, datalink_id_t linkid,
4427     dladm_arg_list_t **listp)
4428 {
4429         dladm_arg_list_t        *list;
4430         dladm_status_t          status = DLADM_STATUS_OK;

4432         list = calloc(1, sizeof (dladm_arg_list_t));
4433         if (list == NULL)
4434                 return (dladm_errno2status(errno));

4436         status = dladm_walk_linkprop(handle, linkid, list,
4437             i_dladm_get_one_prop);

4439         *listp = list;
4440         return (status);
4441 }

4443 /*
4444  * Retrieve the named property from a proplist, check the value and
4445  * convert to a kernel structure.
4446  */
4447 static dladm_status_t
4448 i_dladm_link_proplist_extract_one(dladm_handle_t handle,
4449     dladm_arg_list_t *proplist, const char *name, uint_t flags, void *arg)
4450 {
4451         dladm_status_t          status;
4452         dladm_arg_info_t        *aip = NULL;
4453         int                     i, j;

4455         /* Find named property in proplist */
4456         for (i = 0; i < proplist->al_count; i++) {
4457                 aip = &proplist->al_info[i];
4458                 if (strcasecmp(aip->ai_name, name) == 0)
4459                         break;
4460         }

4462         /* Property not in list */
4463         if (i == proplist->al_count)
4464                 return (DLADM_STATUS_OK);

4466         for (i = 0; i < DLADM_MAX_PROPS; i++) {
4467                 prop_desc_t     *pdp = &prop_table[i];
4468                 val_desc_t      *vdp;
```



```
4470                 vdp = malloc(sizeof (val_desc_t) * aip->ai_count);
4471                 if (vdp == NULL)
4472                         return (DLADM_STATUS_NOMEM);

4474                 if (strcasecmp(aip->ai_name, pdp->pd_name) != 0)
4475                         continue;

4477                 if (aip->ai_val == NULL)
4478                         return (DLADM_STATUS_BADARG);

4480                 /* Check property value */
4481                 if (pdp->pd_check != NULL) {
4482                         status = pdp->pd_check(handle, pdp, 0, aip->ai_val,
4483                             &(aip->ai_count), flags, &vdp, 0);
4484                 } else {
4485                         status = DLADM_STATUS_BADARG;
4486                 }

4488                 if (status != DLADM_STATUS_OK)
4489                         return (status);

4491                 for (j = 0; j < DLADM_MAX_RSRC_PROP; j++) {
4492                         resource_prop_t *rpp = &rsrc_prop_table[j];

4494                         if (strcasecmp(aip->ai_name, rpp->rp_name) != 0)
4495                                 continue;

4497                         /* Extract kernel structure */
4498                         if (rpp->rp_extract != NULL) {
4499                                 status = rpp->rp_extract(vdp,
4500                                     aip->ai_count, arg);
4501                         } else {
4502                                 status = DLADM_STATUS_BADARG;
4503                         }
4504                         break;
4505                 }

4507                 if (status != DLADM_STATUS_OK)
4508                         return (status);

4510                 break;
4511         }
4512         return (status);
4513 }

4515 /*
4516  * Extract properties from a proplist and convert to mac_resource_props_t.
4517  */
4518 dladm_status_t
4519 dladm_link_proplist_extract(dladm_handle_t handle, dladm_arg_list_t *proplist,
4520     mac_resource_props_t *mrp, uint_t flags)
4521 {
4522         dladm_status_t  status;
4523         int             i;

4525         for (i = 0; i < DLADM_MAX_RSRC_PROP; i++) {
4526                 status = i_dladm_link_proplist_extract_one(handle,
4527                     proplist, rsrc_prop_table[i].rp_name, flags, mrp);
4528                 if (status != DLADM_STATUS_OK)
4529                         return (status);
4530         }
4531         return (status);
4532 }

4534 static const char *
4535 dladm_perm2str(uint_t perm, char *buf)
```

```
4536 {
4537         (void) snprintf(buf, DLADM_STRSIZE, "%c%c",
4538             ((perm & MAC_PROP_PERM_READ) != 0) ? 'r' : '-',
4539             ((perm & MAC_PROP_PERM_WRITE) != 0) ? 'w' : '-');
4540         return (buf);
4541 }

4543 dladm_status_t
4544 dladm_get_state(dladm_handle_t handle, datalink_id_t linkid,
4545     link_state_t *state)
4546 {
4547         uint_t                  perms;

4549         return (i_dladm_get_public_prop(handle, linkid, "state", 0,
4550             &perms, state, sizeof (*state)));
4551 }

4553 boolean_t
4554 dladm_attr_is_linkprop(const char *name)
4555 {
4556         /* non-property attribute names */
4557         const char *nonprop[] = {
4558                 /* dlmgmtd core attributes */
4559                 "name",
4560                 "class",
4561                 "media",
4562                 FPHYMAJ,
4563                 FPHYINST,
4564                 FDEVNAME,

4566                 /* other attributes for vlan, aggr, etc */
4567                 DLADM_ATTR_NAMES
4568         };
4569         boolean_t       is_nonprop = B_FALSE;
4570         int             i;

4572         for (i = 0; i < sizeof (nonprop) / sizeof (nonprop[0]); i++) {
4573                 if (strcmp(name, nonprop[i]) == 0) {
4574                         is_nonprop = B_TRUE;
4575                         break;
4576                 }
4577         }

4579         return (!is_nonprop);
4580 }

4582 dladm_status_t
4583 dladm_linkprop_is_set(dladm_handle_t handle, datalink_id_t linkid,
4584     dladm_prop_type_t type, const char *prop_name, boolean_t *is_set)
4585 {
4586         char            *buf, **propvals;
4587         uint_t          valcnt = DLADM_MAX_PROP_VALCNT;
4588         int             i;
4589         dladm_status_t  status = DLADM_STATUS_OK;
4590         size_t          bufsize;

4592         *is_set = B_FALSE;

4594         bufsize = (sizeof (char *) + DLADM_PROP_VAL_MAX) *
4595             DLADM_MAX_PROP_VALCNT;
4596         if ((buf = calloc(1, bufsize)) == NULL)
4597                 return (DLADM_STATUS_NOMEM);

4599         propvals = (char **)(void *)buf;
4600         for (i = 0; i < valcnt; i++) {
4601                 propvals[i] = buf +
```



```
4602                     sizeof (char *) * DLADM_MAX_PROP_VALCNT +
4603                     i * DLADM_PROP_VAL_MAX;
4604         }

4606         if (dladm_get_linkprop(handle, linkid, type, prop_name, propvals,
4607             &valcnt) != DLADM_STATUS_OK) {
4608                 goto done;
4609         }

4611         /*
4612          * valcnt is always set to 1 by get_pool(), hence we need to check
4613          * for a non-null string to see if it is set. For protection and
4614          * allowed-ips, we can check either the *propval or the valcnt.
4615          */
4616         if ((strcmp(prop_name, "pool") == 0 ||
4617             strcmp(prop_name, "protection") == 0 ||
4618             strcmp(prop_name, "allowed-ips") == 0) &&
4619             (strlen(*propvals) != 0)) {
4620                 *is_set = B_TRUE;
4621         } else if ((strcmp(prop_name, "cpus") == 0) && (valcnt != 0)) {
4622                 *is_set = B_TRUE;
4623         } else if ((strcmp(prop_name, "_softmac") == 0) && (valcnt != 0) &&
4624             (strcmp(propvals[0], "true") == 0)) {
4625                 *is_set = B_TRUE;
4626         }

4628 done:
4629         if (buf != NULL)
4630                 free(buf);
4631         return (status);
4632 }

4634 /* ARGSUSED */
4635 static dladm_status_t
4636 get_linkmode_prop(dladm_handle_t handle, prop_desc_t *pdp,
4637     datalink_id_t linkid, char **prop_val, uint_t *val_cnt,
4638     datalink_media_t media, uint_t flags, uint_t *perm_flags)
4639 {
4640         char                    *s;
4641         uint32_t                v;
4642         dladm_status_t          status;

4644         status = i_dladm_get_public_prop(handle, linkid, pdp->pd_name, flags,
4645             perm_flags, &v, sizeof (v));
4646         if (status != DLADM_STATUS_OK)
4647                 return (status);

4649         switch (v) {
4650         case DLADM_PART_CM_MODE:
4651                 s = "cm";
4652                 break;
4653         case DLADM_PART_UD_MODE:
4654                 s = "ud";
4655                 break;
4656         default:
4657                 s = "";
4658                 break;
4659         }
4660         (void) snprintf(prop_val[0], DLADM_STRSIZE, "%s", s);

4662         *val_cnt = 1;
4663         return (DLADM_STATUS_OK);
4664 }
```

```
**********************************************************
  110682 Thu Feb 20 18:59:05 2014
new/usr/src/man/man1m/dladm.1m
2553 mac address should be a dladm link property
**********************************************************
'\" te
.\" Copyright (c) 2008, Sun Microsystems, Inc. All Rights Reserved
.\" Sun Microsystems, Inc. gratefully acknowledges The Open Group for permission
.\" The Institute of Electrical and Electronics Engineers and The Open Group, ha
.\" are reprinted and reproduced in electronic form in the Sun OS Reference Manu
.\" and Electronics Engineers, Inc and The Open Group. In the event of any discr
.\"  This notice shall appear on any product containing this material.
.\" The contents of this file are subject to the terms of the Common Development
.\" See the License for the specific language governing permissions and limitati
.\" fields enclosed by brackets "[]" replaced with your own identifying informat
.TH DLADM 1M "Feb 22, 2014"
.TH DLADM 1M "Sep 23, 2009"
.SH NAME
dladm \- administer data links
.SH SYNOPSIS
.LP
.nf
\fBdladm show-link\fR [\fB-P\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-p\fR
\fBdladm rename-link\fR [\fB-R\fR \fIroot-dir\fR] \fIlink\fR \fInew-link\fR
.fi

.LP
.nf
\fBdladm delete-phys\fR \fIphys-link\fR
\fBdladm show-phys\fR [\fB-P\fR] [\fB-m\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,..
.fi

.LP
.nf
\fBdladm create-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR \fIpolicy
     [\fB-T\fR \fItime\fR] [\fB-u\fR \fIaddress\fR] \fB-l\fR \fIether-link1\fR [
\fBdladm modify-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR \fIpolicy
     [\fB-T\fR \fItime\fR] [\fB-u\fR \fIaddress\fR] \fIaggr-link\fR
\fBdladm delete-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIaggr-link\fR
\fBdladm add-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-link
     \fIaggr-link\fR
\fBdladm remove-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-l
     \fIaggr-link\fR
\fBdladm show-aggr\fR [\fB-PLx\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-p\
     [\fIaggr-link\fR]
.fi

.LP
.nf
\fBdladm create-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\f
     [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-d
     [\fB-l\fR \fIlink\fR...] \fIbridge-name\fR
.fi

.LP
.nf
\fBdladm modify-bridge\fR [\fB-P\fR \fIprotect\fR] [\fB-R\fR \fIroot-dir\fR] [\f
     [\fB-m\fR \fImax-age\fR] [\fB-h\fR \fIhello-time\fR] [\fB-d\fR \fIforward-d
     \fIbridge-name\fR
.fi

.LP
.nf
\fBdladm delete-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fIbridge-name\fR
.fi
```



```
.LP
.nf
\fBdladm add-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\fR \
.fi

.LP
.nf
\fBdladm remove-bridge\fR [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIlink\fR [\fB-l\f
.fi

.LP
.nf
\fBdladm show-bridge\fR [\fB-flt\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]] [[\fB-
     [\fIbridge-name\fR]
.fi

.LP
.nf
\fBdladm create-vlan\fR [\fB-ft\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR \fIether-
\fBdladm delete-vlan\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIvlan-link\fR
\fBdladm show-vlan\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIvla
.fi

.LP
.nf
\fBdladm scan-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIwifi-link\fR]
\fBdladm connect-wifi\fR [\fB-e\fR \fIessid\fR] [\fB-i\fR \fIbssid\fR] [\fB-k\fR
     [\fB-s\fR none | wep | wpa ] [\fB-a\fR open | shared] [\fB-b\fR bss | ibss]
     [\fB-m\fR a | b | g] [\fB-T\fR \fItime\fR] [\fIwifi-link\fR]
\fBdladm disconnect-wifi\fR [\fB-a\fR] [\fIwifi-link\fR]
\fBdladm show-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIwifi-link\fR]
.fi

.LP
.nf
\fBdladm show-ether\fR [\fB-x\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIet
.fi

.LP
.nf
\fBdladm set-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-p\fR \fIprop\f
\fBdladm reset-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-p\fR \fIpro
\fBdladm show-linkprop\fR [\fB-P\fR] [[\fB-c\fR] \fB-o\fR \fIfield\fR[,...]] [\f
.fi

.LP
.nf
\fBdladm create-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-f\fR \fIfile
\fBdladm delete-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fIsecobj\fR[,...
\fBdladm show-secobj\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIs
.fi

.LP
.nf
\fBdladm create-vnic\fR [\fB-t\fR] \fB-l\fR \fIlink\fR [\fB-R\fR \fIroot-dir\fR]
     {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-li
\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

.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.
```

```
391 .RE

393 .sp
394 .ne 2
395 .na
396 \fB\fBvnic\fR\fR
397 .ad
398 .sp .6
399 .RS 4n
400 A virtual network interface. The \fBshow-vnic\fR subcommand displays more
401 detail for this class of datalink.
402 .RE

404 .RE

406 .sp
407 .ne 2
408 .na
409 \fB\fBMTU\fR\fR
410 .ad
411 .sp .6
412 .RS 4n
413 The maximum transmission unit size for the datalink being displayed.
414 .RE

416 .sp
417 .ne 2
418 .na
419 \fB\fBSTATE\fR\fR
420 .ad
421 .sp .6
422 .RS 4n
423 The link state of the datalink. The state can be \fBup\fR, \fBdown\fR, or
424 \fBunknown\fR.
425 .RE

427 .sp
428 .ne 2
429 .na
430 \fB\fBBRIDGE\fR\fR
431 .ad
432 .sp .6
433 .RS 4n
434 The name of the bridge to which this link is assigned, if any.
435 .RE

437 .sp
438 .ne 2
439 .na
440 \fB\fBOVER\fR\fR
441 .ad
442 .sp .6
443 .RS 4n
444 The physical datalink(s) over which the datalink is operating. This applies to
445 \fBaggr\fR, \fBbridge\fR, and \fBvlan\fR classes of datalinks. A VLAN is
446 created over a single physical datalink, a bridge has multiple attached links,
447 and an aggregation is comprised of one or more physical datalinks.
448 .RE

450 When the \fB-o\fR option is used in conjunction with the \fB-s\fR option, used
451 to display link statistics, the field name must be one of the fields listed
452 below, or the special value \fBall\fR to display all fields
453 .sp
454 .ne 2
455 .na
456 \fB\fBLINK\fR\fR
```



```
457 .ad
458 .sp .6
459 .RS 4n
460 The name of the datalink.
461 .RE

463 .sp
464 .ne 2
465 .na
466 \fB\fBIPACKETS\fR\fR
467 .ad
468 .sp .6
469 .RS 4n
470 Number of packets received on this link.
471 .RE

473 .sp
474 .ne 2
475 .na
476 \fB\fBRBYTES\fR\fR
477 .ad
478 .sp .6
479 .RS 4n
480 Number of bytes received on this link.
481 .RE

483 .sp
484 .ne 2
485 .na
486 \fB\fBIERRORS\fR\fR
487 .ad
488 .sp .6
489 .RS 4n
490 Number of input errors.
491 .RE

493 .sp
494 .ne 2
495 .na
496 \fB\fBOPACKETS\fR\fR
497 .ad
498 .sp .6
499 .RS 4n
500 Number of packets sent on this link.
501 .RE

503 .sp
504 .ne 2
505 .na
506 \fB\fBOBYTES\fR\fR
507 .ad
508 .sp .6
509 .RS 4n
510 Number of bytes received on this link.
511 .RE

513 .sp
514 .ne 2
515 .na
516 \fB\fBOERRORS\fR\fR
517 .ad
518 .sp .6
519 .RS 4n
520 Number of output errors.
521 .RE
```

```
523 .RE

525 .sp
526 .ne 2
527 .na
528 \fB\fB-p\fR, \fB--parseable\fR\fR
529 .ad
530 .sp .6
531 .RS 4n
532 Display using a stable machine-parseable format. The \fB-o\fR option is
533 required with \fB-p\fR. See "Parseable Output Format", below.
534 .RE

536 .sp
537 .ne 2
538 .na
539 \fB\fB-P\fR, \fB--persistent\fR\fR
540 .ad
541 .sp .6
542 .RS 4n
543 Display the persistent link configuration.
544 .RE

546 .sp
547 .ne 2
548 .na
549 \fB\fB-s\fR, \fB--statistics\fR\fR
550 .ad
551 .sp .6
552 .RS 4n
553 Display link statistics.
554 .RE

556 .sp
557 .ne 2
558 .na
559 \fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
560 .ad
561 .sp .6
562 .RS 4n
563 Used with the \fB-s\fR option to specify an interval, in seconds, at which
564 statistics should be displayed. If this option is not specified, statistics
565 will be displayed only once.
566 .RE

568 .RE

570 .sp
571 .ne 2
572 .na
573 \fB\fBdladm rename-link\fR [\fB-R\fR \fIroot-dir\fR] \fIlink\fR
574 \fInew-link\fR\fR
575 .ad
576 .sp .6
577 .RS 4n
578 Rename \fIlink\fR to \fInew-link\fR. This is used to give a link a meaningful
579 name, or to associate existing link configuration such as link properties of a
580 removed device with a new device. See the \fBEXAMPLES\fR section for specific
581 examples of how this subcommand is used.
582 .sp
583 .ne 2
584 .na
585 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
586 .ad
587 .sp .6
588 .RS 4n
```



```
589 See "Options," above.
590 .RE

592 .RE

594 .sp
595 .ne 2
596 .na
597 \fB\fBdladm delete-phys\fR \fIphys-link\fR\fR
598 .ad
599 .sp .6
600 .RS 4n
601 This command is used to delete the persistent configuration of a link
602 associated with physical hardware which has been removed from the system. See
603 the \fBEXAMPLES\fR section.
604 .RE

606 .sp
607 .ne 2
608 .na
609 \fB\fBdladm show-phys\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
610 [\fB-H\fR] [\fIphys-link\fR]\fR
611 .ad
612 .sp .6
613 .RS 4n
614 Show the physical device and attributes of all physical links, or of the named
615 physical link. Without \fB-P\fR, only physical links that are available on the
616 running system are displayed.
617 .sp
618 .ne 2
619 .na
620 \fB\fB-H\fR\fR
621 .ad
622 .sp .6
623 .RS 4n
624 Show hardware resource usage, as returned by the NIC driver. Output from
625 \fB-H\fR displays the following elements:
626 .sp
627 .ne 2
628 .na
629 \fB\fBLINK\fR\fR
630 .ad
631 .sp .6
632 .RS 4n
633 A physical device corresponding to a NIC driver.
634 .RE

636 .sp
637 .ne 2
638 .na
639 \fB\fBGROUP\fR\fR
640 .ad
641 .sp .6
642 .RS 4n
643 A collection of rings.
644 .RE

646 .sp
647 .ne 2
648 .na
649 \fB\fBGROUPTYPE\fR\fR
650 .ad
651 .sp .6
652 .RS 4n
653 RX or TX. All rings in a group are of the same group type.
654 .RE
```

```
.sp
.ne 2
.na
\fB\fBRINGS\fR\fR
.ad
.sp .6
.RS 4n
A hardware resource used by a data link, subject to assignment by a driver to
different groups.
.RE

.sp
.ne 2
.na
\fB\fBCLIENTS\fR\fR
.ad
.sp .6
.RS 4n
MAC clients that are using the rings within a group.
.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 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\fBMEDIA\fR\fR
.ad
.sp .6
.RS 4n
The media type provided by the physical datalink.
.RE

.sp
.ne 2
.na
\fB\fBSTATE\fR\fR
.ad
.sp .6
.RS 4n
The state of the link. This can be \fBup\fR, \fBdown\fR, or \fBunknown\fR.
.RE

.sp
.ne 2
```



```
.na
\fB\fBSPEED\fR\fR
.ad
.sp .6
.RS 4n
The current speed of the link, in megabits per second.
.RE

.sp
.ne 2
.na
\fB\fBDUPLEX\fR\fR
.ad
.sp .6
.RS 4n
For Ethernet links, the full/half duplex status of the link is displayed if the
link state is \fBup\fR. The duplex is displayed as \fBunknown\fR in all other
cases.
.RE

.sp
.ne 2
.na
\fB\fBDEVICE\fR\fR
.ad
.sp .6
.RS 4n
The name of the physical device under this link.
.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
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
```

```
 919 .RS 4n
 920 Specifies whether \fBLACP\fR should be used and, if used, the mode in which it
 921 should operate. Supported values are \fBoff\fR, \fBactive\fR or \fBpassive\fR.
 922 .RE

 924 .sp
 925 .ne 2
 926 .na
 927 \fB\fB-T\fR \fItime\fR, \fB--lacp-timer\fR=\fItime\fR\fR
 928 .ad
 929 .br
 930 .na
 931 \fB\fR
 932 .ad
 933 .sp .6
 934 .RS 4n
 935 Specifies the \fBLACP\fR timer value. The supported values are \fBshort\fR or
 936 \fBlong\fRjjj.
 937 .RE

 939 .sp
 940 .ne 2
 941 .na
 942 \fB\fB-u\fR \fIaddress\fR, \fB--unicast\fR=\fIaddress\fR\fR
 943 .ad
 944 .sp .6
 945 .RS 4n
 946 Specifies a fixed unicast hardware address to be used for the aggregation. If
 947 this option is not specified, then an address is automatically chosen from the
 948 set of addresses of the component devices.
 949 .RE

 951 .RE

 953 .sp
 954 .ne 2
 955 .na
 956 \fB\fBdladm modify-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-P\fR
 957 \fIpolicy\fR] [\fB-L\fR \fImode\fR] [\fB-T\fR \fItime\fR] [\fB-u\fR
 958 \fIaddress\fR] \fIaggr-link\fR\fR
 959 .ad
 960 .sp .6
 961 .RS 4n
 962 Modify the parameters of the specified aggregation.
 963 .sp
 964 .ne 2
 965 .na
 966 \fB\fB-t\fR, \fB--temporary\fR\fR
 967 .ad
 968 .sp .6
 969 .RS 4n
 970 Specifies that the modification is temporary. Temporary aggregations last until
 971 the next reboot.
 972 .RE

 974 .sp
 975 .ne 2
 976 .na
 977 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
 978 .ad
 979 .sp .6
 980 .RS 4n
 981 See "Options," above.
 982 .RE

 984 .sp
```



```
 985 .ne 2
 986 .na
 987 \fB\fB-P\fR \fIpolicy\fR, \fB--policy\fR=\fIpolicy\fR\fR
 988 .ad
 989 .sp .6
 990 .RS 4n
 991 Specifies the port selection policy to use for load spreading of outbound
 992 traffic. See \fBdladm create-aggr\fR for a description of valid policy values.
 993 .RE

 995 .sp
 996 .ne 2
 997 .na
 998 \fB\fB-L\fR \fImode\fR, \fB--lacp-mode\fR=\fImode\fR\fR
 999 .ad
1000 .sp .6
1001 .RS 4n
1002 Specifies whether \fBLACP\fR should be used and, if used, the mode in which it
1003 should operate. Supported values are \fBoff\fR, \fBactive\fR, or \fBpassive\fR.
1004 .RE

1006 .sp
1007 .ne 2
1008 .na
1009 \fB\fB-T\fR \fItime\fR, \fB--lacp-timer\fR=\fItime\fR\fR
1010 .ad
1011 .br
1012 .na
1013 \fB\fR
1014 .ad
1015 .sp .6
1016 .RS 4n
1017 Specifies the \fBLACP\fR timer value. The supported values are \fBshort\fR or
1018 \fBlong\fR.
1019 .RE

1021 .sp
1022 .ne 2
1023 .na
1024 \fB\fB-u\fR \fIaddress\fR, \fB--unicast\fR=\fIaddress\fR\fR
1025 .ad
1026 .sp .6
1027 .RS 4n
1028 Specifies a fixed unicast hardware address to be used for the aggregation. If
1029 this option is not specified, then an address is automatically chosen from the
1030 set of addresses of the component devices.
1031 .RE

1033 .RE

1035 .sp
1036 .ne 2
1037 .na
1038 \fB\fBdladm delete-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
1039 \fIaggr-link\fR\fR
1040 .ad
1041 .sp .6
1042 .RS 4n
1043 Deletes the specified aggregation.
1044 .sp
1045 .ne 2
1046 .na
1047 \fB\fB-t\fR, \fB--temporary\fR\fR
1048 .ad
1049 .sp .6
1050 .RS 4n
```

```
1051 Specifies that the deletion is temporary. Temporary deletions last until the
1052 next reboot.
1053 .RE

1055 .sp
1056 .ne 2
1057 .na
1058 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
1059 .ad
1060 .sp .6
1061 .RS 4n
1062 See "Options," above.
1063 .RE

1065 .RE

1067 .sp
1068 .ne 2
1069 .na
1070 \fB\fBdladm add-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
1071 \fIether-link1\fR [\fB--link\fR=\fIether-link2\fR...] \fIaggr-link\fR\fR
1072 .ad
1073 .sp .6
1074 .RS 4n
1075 Adds links to the specified aggregation.
1076 .sp
1077 .ne 2
1078 .na
1079 \fB\fB-l\fR \fIether-link\fR, \fB--link\fR=\fIether-link\fR\fR
1080 .ad
1081 .sp .6
1082 .RS 4n
1083 Specifies an Ethernet link to add to the aggregation. Multiple links can be
1084 added by supplying multiple \fB-l\fR options.
1085 .RE

1087 .sp
1088 .ne 2
1089 .na
1090 \fB\fB-t\fR, \fB--temporary\fR\fR
1091 .ad
1092 .sp .6
1093 .RS 4n
1094 Specifies that the additions are temporary. Temporary additions last until the
1095 next reboot.
1096 .RE

1098 .sp
1099 .ne 2
1100 .na
1101 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
1102 .ad
1103 .sp .6
1104 .RS 4n
1105 See "Options," above.
1106 .RE

1108 .RE

1110 .sp
1111 .ne 2
1112 .na
1113 \fB\fBdladm remove-aggr\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
1114 \fIether-link1\fR [\fB--l\fR=\fIether-link2\fR...] \fIaggr-link\fR\fR
1115 .ad
1116 .sp .6
```



```
1117 .RS 4n
1118 Removes links from the specified aggregation.
1119 .sp
1120 .ne 2
1121 .na
1122 \fB\fB-l\fR \fIether-link\fR, \fB--link\fR=\fIether-link\fR\fR
1123 .ad
1124 .sp .6
1125 .RS 4n
1126 Specifies an Ethernet link to remove from the aggregation. Multiple links can
1127 be added by supplying multiple \fB-l\fR options.
1128 .RE

1130 .sp
1131 .ne 2
1132 .na
1133 \fB\fB-t\fR, \fB--temporary\fR\fR
1134 .ad
1135 .sp .6
1136 .RS 4n
1137 Specifies that the removals are temporary. Temporary removal last until the
1138 next reboot.
1139 .RE

1141 .sp
1142 .ne 2
1143 .na
1144 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
1145 .ad
1146 .sp .6
1147 .RS 4n
1148 See "Options," above.
1149 .RE

1151 .RE

1153 .sp
1154 .ne 2
1155 .na
1156 \fB\fBdladm show-aggr\fR [\fB-PLx\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
1157 [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]] [\fIaggr-link\fR]\fR
1158 .ad
1159 .sp .6
1160 .RS 4n
1161 Show aggregation configuration (the default), \fBLACP\fR information, or
1162 statistics, either for all aggregations or for the specified aggregation.
1163 .sp
1164 By default (with no options), the following fields can be displayed:
1165 .sp
1166 .ne 2
1167 .na
1168 \fB\fBLINK\fR\fR
1169 .ad
1170 .sp .6
1171 .RS 4n
1172 The name of the aggregation link.
1173 .RE

1175 .sp
1176 .ne 2
1177 .na
1178 \fB\fBPOLICY\fR\fR
1179 .ad
1180 .sp .6
1181 .RS 4n
1182 The LACP policy of the aggregation. See the \fBcreate-aggr\fR \fB-P\fR option
```

```
1183 for a description of the possible values.
1184 .RE

1186 .sp
1187 .ne 2
1188 .na
1189 \fB\fBADDRPOLICY\fR\fR
1190 .ad
1191 .sp .6
1192 .RS 4n
1193 Either \fBauto\fR, if the aggregation is configured to automatically configure
1194 its unicast MAC address (the default if the \fB-u\fR option was not used to
1195 create or modify the aggregation), or \fBfixed\fR, if \fB-u\fR was used to set
1196 a fixed MAC address.
1197 .RE

1199 .sp
1200 .ne 2
1201 .na
1202 \fB\fBLACPACTIVITY\fR\fR
1203 .ad
1204 .sp .6
1205 .RS 4n
1206 The LACP mode of the aggregation. Possible values are \fBoff\fR, \fBactive\fR,
1207 or \fBpassive\fR, as set by the \fB-l\fR option to \fBcreate-aggr\fR or
1208 \fBmodify-aggr\fR.
1209 .RE

1211 .sp
1212 .ne 2
1213 .na
1214 \fB\fBLACPTIMER\fR\fR
1215 .ad
1216 .sp .6
1217 .RS 4n
1218 The LACP timer value of the aggregation as set by the \fB-T\fR option of
1219 \fBcreate-aggr\fR or \fBmodify-aggr\fR.
1220 .RE

1222 .sp
1223 .ne 2
1224 .na
1225 \fB\fBFLAGS\fR\fR
1226 .ad
1227 .sp .6
1228 .RS 4n
1229 A set of state flags associated with the aggregation. The only possible flag is
1230 \fBf\fR, which is displayed if the administrator forced the creation the
1231 aggregation using the \fB-f\fR option to \fBcreate-aggr\fR. Other flags might
1232 be defined in the future.
1233 .RE

1235 The \fBshow-aggr\fR command accepts the following options:
1236 .sp
1237 .ne 2
1238 .na
1239 \fB\fB-L\fR, \fB--lacp\fR\fR
1240 .ad
1241 .sp .6
1242 .RS 4n
1243 Displays detailed \fBLACP\fR information for the aggregation link and each
1244 underlying port. Most of the state information displayed by this option is
1245 defined by IEEE 802.3. With this option, the following fields can be displayed:
1246 .sp
1247 .ne 2
1248 .na
```



```
1249 \fB\fBLINK\fR\fR
1250 .ad
1251 .sp .6
1252 .RS 4n
1253 The name of the aggregation link.
1254 .RE

1256 .sp
1257 .ne 2
1258 .na
1259 \fB\fBPORT\fR\fR
1260 .ad
1261 .sp .6
1262 .RS 4n
1263 The name of one of the underlying aggregation ports.
1264 .RE

1266 .sp
1267 .ne 2
1268 .na
1269 \fB\fBAGGREGATABLE\fR\fR
1270 .ad
1271 .sp .6
1272 .RS 4n
1273 Whether the port can be added to the aggregation.
1274 .RE

1276 .sp
1277 .ne 2
1278 .na
1279 \fB\fBSYNC\fR\fR
1280 .ad
1281 .sp .6
1282 .RS 4n
1283 If \fByes\fR, the system considers the port to be synchronized and part of the
1284 aggregation.
1285 .RE

1287 .sp
1288 .ne 2
1289 .na
1290 \fB\fBCOLL\fR\fR
1291 .ad
1292 .sp .6
1293 .RS 4n
1294 If \fByes\fR, collection of incoming frames is enabled on the associated port.
1295 .RE

1297 .sp
1298 .ne 2
1299 .na
1300 \fB\fBDIST\fR\fR
1301 .ad
1302 .sp .6
1303 .RS 4n
1304 If \fByes\fR, distribution of outgoing frames is enabled on the associated
1305 port.
1306 .RE

1308 .sp
1309 .ne 2
1310 .na
1311 \fB\fBDEFAULTED\fR\fR
1312 .ad
1313 .sp .6
1314 .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]]
```

```
1711 [[\fB-p\fR] \fB-o\fR \fIfield\fR,...] [\fIbridge-name\fR]\fR
1712 .ad
1713 .sp .6
1714 .RS 4n
1715 Show the running status and configuration of bridges, their attached links,
1716 learned forwarding entries, and \fBTRILL\fR nickname databases. When showing
1717 overall bridge status and configuration, the bridge name can be omitted to show
1718 all bridges. The other forms require a specified bridge.
1719 .sp
1720 The show-bridge subcommand accepts the following options:
1721 .sp
1722 .ne 2
1723 .na
1724 \fB\fB-i\fR \fIinterval\fR, \fB--interval\fR=\fIinterval\fR\fR
1725 .ad
1726 .sp .6
1727 .RS 4n
1728 Used with the \fB-s\fR option to specify an interval, in seconds, at which
1729 statistics should be displayed. If this option is not specified, statistics
1730 will be displayed only once.
1731 .RE

1733 .sp
1734 .ne 2
1735 .na
1736 \fB\fB-s\fR, \fB--statistics\fR\fR
1737 .ad
1738 .sp .6
1739 .RS 4n
1740 Display statistics for the specified bridges or for a given bridge’s attached
1741 links. This option cannot be used with the \fB-f\fR and \fB-t\fR options.
1742 .RE

1744 .sp
1745 .ne 2
1746 .na
1747 \fB\fB-p\fR, \fB--parseable\fR\fR
1748 .ad
1749 .sp .6
1750 .RS 4n
1751 Display using a stable machine-parsable format. See "Parsable Output Format,"
1752 below.
1753 .RE

1755 .sp
1756 .ne 2
1757 .na
1758 \fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
1759 .ad
1760 .sp .6
1761 .RS 4n
1762 A case-insensitive, comma-separated list of output fields to display. The field
1763 names are described below. The special value all displays all fields. Each set
1764 of fields has its own default set to display when \fB-o\fR is not specified.
1765 .RE

1767 By default, the \fBshow-bridge\fR subcommand shows bridge configuration. The
1768 following fields can be shown:
1769 .sp
1770 .ne 2
1771 .na
1772 \fB\fBBRIDGE\fR\fR
1773 .ad
1774 .sp .6
1775 .RS 4n
1776 The name of the bridge.
```



```
1777 .RE

1779 .sp
1780 .ne 2
1781 .na
1782 \fB\fBADDRESS\fR\fR
1783 .ad
1784 .sp .6
1785 .RS 4n
1786 The Bridge Unique Identifier value (MAC address).
1787 .RE

1789 .sp
1790 .ne 2
1791 .na
1792 \fB\fBPRIORITY\fR\fR
1793 .ad
1794 .sp .6
1795 .RS 4n
1796 Configured priority value; set by \fB-p\fR with \fBcreate-bridge\fR and
1797 \fBmodify-bridge\fR.
1798 .RE

1800 .sp
1801 .ne 2
1802 .na
1803 \fB\fBBMAXAGE\fR\fR
1804 .ad
1805 .sp .6
1806 .RS 4n
1807 Configured bridge maximum age; set by \fB-m\fR with \fBcreate-bridge\fR and
1808 \fBmodify-bridge\fR.
1809 .RE

1811 .sp
1812 .ne 2
1813 .na
1814 \fB\fBHELLOTIME\fR\fR
1815 .ad
1816 .sp .6
1817 .RS 4n
1818 Configured bridge hello time; set by \fB-h\fR with \fBcreate-bridge\fR and
1819 \fBmodify-bridge\fR.
1820 .RE

1822 .sp
1823 .ne 2
1824 .na
1825 \fB\fBFWDDELAY\fR\fR
1826 .ad
1827 .sp .6
1828 .RS 4n
1829 Configured forwarding delay; set by \fB-d\fR with \fBcreate-bridge\fR and
1830 \fBmodify-bridge\fR.
1831 .RE

1833 .sp
1834 .ne 2
1835 .na
1836 \fB\fBFORCEPROTO\fR\fR
1837 .ad
1838 .sp .6
1839 .RS 4n
1840 Configured forced maximum protocol; set by \fB-f\fR with \fBcreate-bridge\fR
1841 and \fBmodify-bridge\fR.
1842 .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
This indicates whether point-to-point (\fBP2P\fR) mode been detected.
.RE

.sp
.ne 2
.na
\fB\fBOPEREDGE\fR\fR
.ad
.sp .6
.RS 4n
This indicates whether edge mode has been detected.
.RE

.sp
.ne 2
```

```
2107 .na
2108 \fB\fBDESROOT\fR\fR
2109 .ad
2110 .sp .6
2111 .RS 4n
2112 The Root Bridge Identifier that has been seen on this port.
2113 .RE

2115 .sp
2116 .ne 2
2117 .na
2118 \fB\fBDESCOST\fR\fR
2119 .ad
2120 .sp .6
2121 .RS 4n
2122 Path cost to the network root node through the designated port.
2123 .RE

2125 .sp
2126 .ne 2
2127 .na
2128 \fB\fBDESBRIDGE\fR\fR
2129 .ad
2130 .sp .6
2131 .RS 4n
2132 Bridge Identifier for this port.
2133 .RE

2135 .sp
2136 .ne 2
2137 .na
2138 \fB\fBDESPORT\fR\fR
2139 .ad
2140 .sp .6
2141 .RS 4n
2142 The ID and priority of the port used to transmit configuration messages for
2143 this port.
2144 .RE

2146 .sp
2147 .ne 2
2148 .na
2149 \fB\fBTCACK\fR\fR
2150 .ad
2151 .sp .6
2152 .RS 4n
2153 This indicates whether Topology Change Acknowledge has been seen.
2154 .RE

2156 When the \fB-l\fR option is specified without the \fB-o\fR option, only the
2157 \fBLINK\fR, \fBSTATE\fR, \fBUPTIME\fR, and \fBDESROOT\fR fields are shown.
2158 .sp
2159 When the \fB-l\fR option is specified, the \fB-s\fR option can be used to
2160 display the following fields for each link:
2161 .sp
2162 .ne 2
2163 .na
2164 \fB\fBLINK\fR\fR
2165 .ad
2166 .sp .6
2167 .RS 4n
2168 Link name.
2169 .RE

2171 .sp
2172 .ne 2
```



```
2173 .na
2174 \fB\fBCFGBPDU\fR\fR
2175 .ad
2176 .sp .6
2177 .RS 4n
2178 Number of configuration BPDUs received.
2179 .RE

2181 .sp
2182 .ne 2
2183 .na
2184 \fB\fBTCNBPDU\fR\fR
2185 .ad
2186 .sp .6
2187 .RS 4n
2188 Number of topology change BPDUs received.
2189 .RE

2191 .sp
2192 .ne 2
2193 .na
2194 \fB\fBRSTPBPDU\fR\fR
2195 .ad
2196 .sp .6
2197 .RS 4n
2198 Number of Rapid Spanning Tree BPDUs received.
2199 .RE

2201 .sp
2202 .ne 2
2203 .na
2204 \fB\fBTXBPDU\fR\fR
2205 .ad
2206 .sp .6
2207 .RS 4n
2208 Number of BPDUs transmitted.
2209 .RE

2211 .sp
2212 .ne 2
2213 .na
2214 \fB\fBDROPS\fR\fR
2215 .ad
2216 .sp .6
2217 .RS 4n
2218 Number of packets dropped due to resource problems.
2219 .RE

2221 .sp
2222 .ne 2
2223 .na
2224 \fB\fBRECV\fR\fR
2225 .ad
2226 .sp .6
2227 .RS 4n
2228 Number of packets received by the bridge.
2229 .RE

2231 .sp
2232 .ne 2
2233 .na
2234 \fB\fBXMIT\fR\fR
2235 .ad
2236 .sp .6
2237 .RS 4n
2238 Number of packets sent by the bridge.
```

```
.RE

When the \fB-o\fR option is not specified, only the \fBLINK\fR, \fBDROPS\fR,
\fBRECV\fR, and \fBXMIT\fR fields are shown.
.RE

.sp
.ne 2
.na
\fB\fB-f\fR, \fB--forwarding\fR\fR
.ad
.sp .6
.RS 4n
Displays forwarding entries for a single bridge instance. With this option, the
following fields can be shown for each forwarding entry:
.sp
.ne 2
.na
\fB\fBDEST\fR\fR
.ad
.sp .6
.RS 4n
Destination MAC address.
.RE

.sp
.ne 2
.na
\fB\fBAGE\fR\fR
.ad
.sp .6
.RS 4n
Age of entry in seconds and milliseconds. Omitted for local entries.
.RE

.sp
.ne 2
.na
\fB\fBFLAGS\fR\fR
.ad
.sp .6
.RS 4n
The \fBL\fR (local) flag is shown if the MAC address belongs to an attached
link or to a VNIC on one of the attached links.
.RE

.sp
.ne 2
.na
\fB\fBOUTPUT\fR\fR
.ad
.sp .6
.RS 4n
For local entries, this is the name of the attached link that has the MAC
address. Otherwise, for bridges that use Spanning Tree Protocol, this is the
output interface name. For RBridges, this is the output \fBTRILL\fR nickname.
.RE

When the \fB-o\fR option is not specified, the \fBDEST\fR, \fBAGE\fR,
\fBFLAGS\fR, and \fBOUTPUT\fR fields are shown.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--trill\fR\fR
```



```
.ad
.sp .6
.RS 4n
Displays \fBTRILL\fR nickname entries for a single bridge instance. With this
option, the following fields can be shown for each \fBTRILL\fR nickname entry:
.sp
.ne 2
.na
\fB\fBNICK\fR\fR
.ad
.sp .6
.RS 4n
\fBTRILL\fR nickname for this RBridge, which is a number from 1 to 65535.
.RE

.sp
.ne 2
.na
\fB\fBFLAGS\fR\fR
.ad
.sp .6
.RS 4n
The \fBL\fR flag is shown if the nickname identifies the local system.
.RE

.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
Link name for output when sending messages to this RBridge.
.RE

.sp
.ne 2
.na
\fB\fBNEXTHOP\fR\fR
.ad
.sp .6
.RS 4n
MAC address of the next hop RBridge that is used to reach the RBridge with this
nickname.
.RE

When the \fB-o\fR option is not specified, the \fBNICK\fR, \fBFLAGS\fR,
\fBLINK\fR, and \fBNEXTHOP\fR fields are shown.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm create-vlan\fR [\fB-ft\fR] [\fB-R\fR \fIroot-dir\fR] \fB-l\fR
\fIether-link\fR \fB-v\fR \fIvid\fR [\fIvlan-link\fR]\fR
.ad
.sp .6
.RS 4n
Create a tagged VLAN link with an ID of \fIvid\fR over Ethernet link
\fIether-link\fR. The name of the VLAN link can be specified as
\fIvlan\fR-\fIlink\fR. If the name is not specified, a name will be
automatically generated (assuming that \fIether-link\fR is \fIname\fR\fIPPA\fR)
as:
.sp
```

```
.in +2
.nf
<\fIname\fR><1000 * \fIvlan-tag\fR + \fIPPA\fR>
.fi
.in -2
.sp

For example, if \fIether-link\fR is \fBbge1\fR and \fIvid\fR is 2, the name
generated is \fBbge2001\fR.
.sp
.ne 2
.na
\fB\fB-f\fR, \fB--force\fR\fR
.ad
.sp .6
.RS 4n
Force the creation of the VLAN link. Some devices do not allow frame sizes
large enough to include a VLAN header. When creating a VLAN link over such a
device, the \fB-f\fR option is needed, and the MTU of the IP interfaces on the
resulting VLAN must be set to 1496 instead of 1500.
.RE

.sp
.ne 2
.na
\fB\fB-l\fR \fIether-link\fR\fR
.ad
.sp .6
.RS 4n
Specifies Ethernet link over which VLAN is created.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the VLAN link is temporary. Temporary VLAN links 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 delete-vlan\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIvlan-link\fR\fR
.ad
.sp .6
.RS 4n
Delete the VLAN link specified.
.sp
The \fBdelete-vlan\fRsubcommand accepts the following options:
```



```
.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-vlan\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
[\fIvlan-link\fR]\fR
.ad
.sp .6
.RS 4n
Display VLAN configuration for all VLAN links or for the specified VLAN link.
.sp
The \fBshow-vlan\fRsubcommand accepts the following options:
.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 VLAN link, the following fields can be displayed:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the VLAN link.
.RE

.sp
.ne 2
.na
\fB\fBVID\fR\fR
.ad
.sp .6
.RS 4n
The ID associated with the VLAN.
.RE

.sp
.ne 2
```

```
.na
\fB\fBOVER\fR\fR
.ad
.sp .6
.RS 4n
The name of the physical link over which this VLAN is configured.
.RE

.sp
.ne 2
.na
\fB\fBFLAGS\fR\fR
.ad
.sp .6
.RS 4n
A set of flags associated with the VLAN link. Possible flags are:
.sp
.ne 2
.na
\fB\fBf\fR\fR
.ad
.sp .6
.RS 4n
The VLAN was created using the \fB-f\fR option to \fBcreate-vlan\fR.
.RE

.sp
.ne 2
.na
\fB\fBi\fR\fR
.ad
.sp .6
.RS 4n
The VLAN was implicitly created when the DLPI link was opened. These VLAN links
are automatically deleted on last close of the DLPI link (for example, when the
IP interface associated with the VLAN link is unplumbed).
.RE

Additional flags might be defined in the future.
.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 VLAN configuration rather than the state of the running
system.
.RE

.RE
```



```
.sp
.ne 2
.na
\fB\fBdladm scan-wifi\fR [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
[\fIwifi-link\fR]\fR
.ad
.sp .6
.RS 4n
Scans for \fBWiFi\fR networks, either on all \fBWiFi\fR links, or just on the
specified \fIwifi-link\fR.
.sp
By default, currently all fields but \fBBSSTYPE\fR 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 \fBWiFi\fR network found, the following fields can
be displayed:
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the link the \fBWiFi\fR network is on.
.RE

.sp
.ne 2
.na
\fB\fBESSID\fR\fR
.ad
.sp .6
.RS 4n
The \fBESSID\fR (name) of the \fBWiFi\fR network.
.RE

.sp
.ne 2
.na
\fB\fBBSSID\fR\fR
.ad
.sp .6
.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
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the link being displayed.
.RE

.sp
.ne 2
.na
\fB\fBSTATUS\fR\fR
.ad
.sp .6
.RS 4n
Either \fBconnected\fR if the link is connected, or \fBdisconnected\fR if it is
```

not connected. If the link is disconnected, all remaining fields have the value \fB--\fR.
.RE

.sp
.ne 2
.na
\fB\fBESSID\fR\fR
.ad
.sp .6
.RS 4n
The \fBESSID\fR (name) of the connected \fBWiFi\fR network.
.RE

.sp
.ne 2
.na
\fB\fBBSSID\fR\fR
.ad
.sp .6
.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, or \fBwpa\fR for a WiFi network that requires WPA.
.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 connection strength: 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 connection speed, in megabits per second.
.RE

.sp
.ne 2
.na
\fB\fBAUTH\fR\fR
.ad
.sp .6
.RS 4n
Either \fBopen\fR or \fBshared\fR (see \fBconnect-wifi\fR).
.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

By default, currently all fields but \fBAUTH\fR, \fBBSSID\fR, \fBBSSTYPE\fR are displayed.
.RE

.sp
.ne 2
.na
\fB\fB-p\fR, \fB--parseable\fR\fR
.ad
.sp .6
.RS 4n
Displays 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 show-ether\fR [\fB-x\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR,...] [\fIether-link\fR]\fR
.ad
.sp .6
.RS 4n
Shows state information either for all physical Ethernet links or for a specified physical Ethernet link.
.sp
The \fBshow-ether\fR subcommand accepts the following options:
.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

```
3031 \fB\fBLINK\fR\fR
3032 .ad
3033 .sp .6
3034 .RS 4n
3035 The name of the link being displayed.
3036 .RE

3038 .sp
3039 .ne 2
3040 .na
3041 \fB\fBPTYPE\fR\fR
3042 .ad
3043 .sp .6
3044 .RS 4n
3045 Parameter type, where \fBcurrent\fR indicates the negotiated state of the link,
3046 \fBcapable\fR indicates capabilities supported by the device, \fBadv\fR
3047 indicates the advertised capabilities, and \fBpeeradv\fR indicates the
3048 capabilities advertised by the link-partner.
3049 .RE

3051 .sp
3052 .ne 2
3053 .na
3054 \fB\fBSTATE\fR\fR
3055 .ad
3056 .sp .6
3057 .RS 4n
3058 The state of the link.
3059 .RE

3061 .sp
3062 .ne 2
3063 .na
3064 \fB\fBAUTO\fR\fR
3065 .ad
3066 .sp .6
3067 .RS 4n
3068 A \fByes\fR/\fBno\fR value indicating whether auto-negotiation is advertised.
3069 .RE

3071 .sp
3072 .ne 2
3073 .na
3074 \fB\fBSPEED-DUPLEX\fR\fR
3075 .ad
3076 .sp .6
3077 .RS 4n
3078 Combinations of speed and duplex values available. The units of speed are
3079 encoded with a trailing suffix of \fBG\fR (Gigabits/s) or \fBM\fR (Mb/s).
3080 Duplex values are encoded as \fBf\fR (full-duplex) or \fBh\fR (half-duplex).
3081 .RE

3083 .sp
3084 .ne 2
3085 .na
3086 \fB\fBPAUSE\fR\fR
3087 .ad
3088 .sp .6
3089 .RS 4n
3090 Flow control information. Can be \fBno\fR, indicating no flow control is
3091 available; \fBtx\fR, indicating that the end-point can transmit pause frames,
3092 but ignores any received pause frames; \fBrx\fR, indicating that the end-point
3093 receives and acts upon received pause frames; or \fBbi\fR, indicating
3094 bi-directional flow-control.
3095 .RE
```



```
3097 .sp
3098 .ne 2
3099 .na
3100 \fB\fBREM_FAULT\fR\fR
3101 .ad
3102 .sp .6
3103 .RS 4n
3104 Fault detection information. Valid values are \fBnone\fR or \fBfault\fR.
3105 .RE

3107 By default, all fields except \fBREM_FAULT\fR are displayed for the "current"
3108 \fBPTYPE\fR.
3109 .RE

3111 .sp
3112 .ne 2
3113 .na
3114 \fB\fB-p\fR, \fB--parseable\fR\fR
3115 .ad
3116 .sp .6
3117 .RS 4n
3118 Displays using a stable machine-parseable format. The \fB-o\fR option is
3119 required with \fB-p\fR. See "Parseable Output Format", below.
3120 .RE

3122 .sp
3123 .ne 2
3124 .na
3125 \fB\fB-x\fR, \fB--extended\fR\fR
3126 .ad
3127 .sp .6
3128 .RS 4n
3129 Extended output is displayed for \fBPTYPE\fR values of \fBcurrent\fR,
3130 \fBcapable\fR, \fBadv\fR and \fBpeeradv\fR.
3131 .RE

3133 .RE

3135 .sp
3136 .ne 2
3137 .na
3138 \fB\fBdladm set-linkprop\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] \fB-p\fR
3139 \fIprop\fR=\fIvalue\fR[,...] \fIlink\fR\fR
3140 .ad
3141 .sp .6
3142 .RS 4n
3143 Sets the values of one or more properties on the link specified. The list of
3144 properties and their possible values depend on the link type, the network
3145 device driver, and networking hardware. These properties can be retrieved using
3146 \fBshow-linkprop\fR.
3147 .sp
3148 .ne 2
3149 .na
3150 \fB\fB-t\fR, \fB--temporary\fR\fR
3151 .ad
3152 .sp .6
3153 .RS 4n
3154 Specifies that the changes are temporary. Temporary changes last until the next
3155 reboot.
3156 .RE

3158 .sp
3159 .ne 2
3160 .na
3161 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
3162 .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
```

```
3295 shown as \fB--\fR. If it is unknown, the value is shown as \fB?\fR. Persistent
3296 values that are not set or have been reset will be shown as \fB--\fR and will
3297 use the system \fBDEFAULT\fR value (if any).
3298 .RE

3300 .sp
3301 .ne 2
3302 .na
3303 \fB\fBDEFAULT\fR\fR
3304 .ad
3305 .sp .6
3306 .RS 4n
3307 The default value of the property. If the property has no default value,
3308 \fB--\fR is shown.
3309 .RE

3311 .sp
3312 .ne 2
3313 .na
3314 \fB\fBPOSSIBLE\fR\fR
3315 .ad
3316 .sp .6
3317 .RS 4n
3318 A comma-separated list of the values the property can have. If the values span
3319 a numeric range, \fImin\fR - \fImax\fR might be shown as shorthand. If the
3320 possible values are unknown or unbounded, \fB--\fR is shown.
3321 .RE

3323 The list of properties depends on the link type and network device driver, and
3324 the available values for a given property further depends on the underlying
3325 network hardware and its state. General link properties are documented in the
3326 \fBLINK PROPERTIES\fR section. However, link properties that begin with
3327 "\fB_\fR" (underbar) are specific to a given link or its underlying network
3328 device and subject to change or removal. See the appropriate network device
3329 driver man page for details.
3330 .RE

3332 .sp
3333 .ne 2
3334 .na
3335 \fB\fB-c\fR, \fB--parseable\fR\fR
3336 .ad
3337 .sp .6
3338 .RS 4n
3339 Display using a stable machine-parseable format. The \fB-o\fR option is
3340 required with this option. See "Parseable Output Format", below.
3341 .RE

3343 .sp
3344 .ne 2
3345 .na
3346 \fB\fB-P\fR, \fB--persistent\fR\fR
3347 .ad
3348 .sp .6
3349 .RS 4n
3350 Display persistent link property information
3351 .RE

3353 .sp
3354 .ne 2
3355 .na
3356 \fB\fB-p\fR \fIprop, ...\fR, \fB--prop\fR=\fIprop, ...\fR\fR
3357 .ad
3358 .sp .6
3359 .RS 4n
3360 A comma-separated list of properties to show. See the sections on link
```



```
3361 properties following subcommand descriptions.
3362 .RE

3364 .RE

3366 .sp
3367 .ne 2
3368 .na
3369 \fB\fBdladm create-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR] [\fB-f\fR
3370 \fIfile\fR] \fB-c\fR \fIclass\fR \fIsecobj\fR\fR
3371 .ad
3372 .sp .6
3373 .RS 4n
3374 Create a secure object named \fIsecobj\fR in the specified \fIclass\fR to be
3375 later used as a WEP or WPA key in connecting to an encrypted network. The value
3376 of the secure object can either be provided interactively or read from a file.
3377 The sequence of interactive prompts and the file format depends on the class of
3378 the secure object.
3379 .sp
3380 Currently, the classes \fBwep\fR and \fBwpa\fR are supported. The \fBWEP\fR
3381 (Wired Equivalent Privacy) key can be either 5 or 13 bytes long. It can be
3382 provided either as an \fBASCII\fR or hexadecimal string -- thus, \fB12345\fR
3383 and \fB0x3132333435\fR are equivalent 5-byte keys (the \fB0x\fR prefix can be
3384 omitted). A file containing a \fBWEP\fR key must consist of a single line using
3385 either \fBWEP\fR key format. The WPA (Wi-Fi Protected Access) key must be
3386 provided as an ASCII string with a length between 8 and 63 bytes.
3387 .sp
3388 This subcommand is only usable by users or roles that belong to the "Network
3389 Link Security" \fBRBAC\fR profile.
3390 .sp
3391 .ne 2
3392 .na
3393 \fB\fB-c\fR \fIclass\fR, \fB--class\fR=\fIclass\fR\fR
3394 .ad
3395 .sp .6
3396 .RS 4n
3397 \fIclass\fR can be \fBwep\fR or \fBwpa\fR. See preceding discussion.
3398 .RE

3400 .sp
3401 .ne 2
3402 .na
3403 \fB\fB-t\fR, \fB--temporary\fR\fR
3404 .ad
3405 .sp .6
3406 .RS 4n
3407 Specifies that the creation is temporary. Temporary creation last until the
3408 next reboot.
3409 .RE

3411 .sp
3412 .ne 2
3413 .na
3414 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
3415 .ad
3416 .sp .6
3417 .RS 4n
3418 See "Options," above.
3419 .RE

3421 .sp
3422 .ne 2
3423 .na
3424 \fB\fB-f\fR \fIfile\fR, \fB--file\fR=\fIfile\fR\fR
3425 .ad
3426 .sp .6
```

```
.RS 4n
Specifies a file that should be used to obtain the secure object’s value. The
format of this file depends on the secure object class. See the \fBEXAMPLES\fR
section for an example of using this option to set a \fBWEP\fR key.
.RE

.RE

.sp
.ne 2
.na
\fB\fBdladm delete-secobj\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIsecobj\fR[,...]\fR
.ad
.sp .6
.RS 4n
Delete one or more specified secure objects. 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-t\fR, \fB--temporary\fR\fR
.ad
.sp .6
.RS 4n
Specifies that the deletions are 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-secobj\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
[\fIsecobj\fR,...]\fR
.ad
.sp .6
.RS 4n
Show current or persistent secure object information. If one or more secure
objects are specified, then information for each is displayed. Otherwise, all
current or persistent secure objects are displayed.
.sp
By default, current secure objects are displayed, which are all secure objects
that have either been persistently created and not temporarily deleted, or
temporarily created.
.sp
For security reasons, it is not possible to show the value of a secure object.
.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. For displayed secure object, the
following fields can be shown:
.sp
.ne 2
.na
\fB\fBOBJECT\fR\fR
.ad
.sp .6
.RS 4n
The name of the secure object.
.RE

.sp
.ne 2
.na
\fB\fBCLASS\fR\fR
.ad
.sp .6
.RS 4n
The class of the secure object.
.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 persistent secure object information
.RE

.RE

.sp
.ne 2
.na
\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
\fBdladm delete-vnic\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
\fIvnic-link\fR\fR
.ad
.sp .6
.RS 4n
Deletes the specified VNIC.
.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-vnic\fR [\fB-pP\fR] [\fB-s\fR [\fB-i\fR \fIinterval\fR]]
[\fB-o\fR \fIfield\fR[,...]] [\fB-l\fR \fIlink\fR] [\fIvnic-link\fR]\fR
.ad
.sp .6
.RS 4n
Show VNIC configuration information (the default) or statistics, for all VNICs,
all VNICs on a link, or only the specified \fIvnic-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. 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-vnic\fR displays all fields.
.sp
.ne 2
.na
\fB\fBLINK\fR\fR
.ad
.sp .6
.RS 4n
The name of the VNIC.
.RE

.sp
.ne 2
.na
\fB\fBOVER\fR\fR
.ad
.sp .6
.RS 4n
The name of the physical link over which this VNIC is configured.
.RE

.sp
.ne 2
.na
\fB\fBSPEED\fR\fR
.ad
.sp .6
.RS 4n
The maximum speed of the VNIC, in megabits per second.
.RE

.sp
.ne 2
.na
\fB\fBMACADDRESS\fR\fR
```



```
.ad
.sp .6
.RS 4n
MAC address of the VNIC.
.RE

.sp
.ne 2
.na
\fB\fBMACADDRTYPE\fR\fR
.ad
.sp .6
.RS 4n
MAC address type of the VNIC. \fBdladm\fR distinguishes among the following MAC
address types:
.sp
.ne 2
.na
\fB\fBrandom\fR\fR
.ad
.sp .6
.RS 4n
A random address assigned to the VNIC.
.RE

.sp
.ne 2
.na
\fB\fBfactory\fR\fR
.ad
.sp .6
.RS 4n
A factory MAC address used by the VNIC.
.RE

.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 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

```
4087 description.
4088 .RE

4090 .sp
4091 .ne 2
4092 .na
4093 \fB\fB-d\fR \fItdst\fR, \fB--tunnel-dst\fR=\fItdst\fR\fR
4094 .ad
4095 .sp .6
4096 .RS 4n
4097 Specifies a new tunnel destination address. See \fBcreate-iptun\fR for a
4098 description.
4099 .RE

4101 .RE

4103 .sp
4104 .ne 2
4105 .na
4106 \fB\fBdladm delete-iptun\fR [\fB-t\fR] [\fB-R\fR \fIroot-dir\fR]
4107 \fIiptun-link\fR\fR
4108 .ad
4109 .sp .6
4110 .RS 4n
4111 Delete the specified IP tunnel link.
4112 .sp
4113 .ne 2
4114 .na
4115 \fB\fB-t\fR, \fB--temporary\fR\fR
4116 .ad
4117 .sp .6
4118 .RS 4n
4119 Specifies that the deletion is temporary. Temporary deletions last until the
4120 next reboot.
4121 .RE

4123 .sp
4124 .ne 2
4125 .na
4126 \fB\fB-R\fR \fIroot-dir\fR, \fB--root-dir\fR=\fIroot-dir\fR\fR
4127 .ad
4128 .sp .6
4129 .RS 4n
4130 See "Options," above.
4131 .RE

4133 .RE

4135 .sp
4136 .ne 2
4137 .na
4138 \fB\fBdladm show-iptun\fR [\fB-P\fR] [[\fB-p\fR] \fB-o\fR \fIfield\fR[,...]]
4139 [\fIiptun-link\fR]\fR
4140 .ad
4141 .sp .6
4142 .RS 4n
4143 Show IP tunnel link configuration for a single IP tunnel or all IP tunnels.
4144 .sp
4145 .ne 2
4146 .na
4147 \fB\fB-P\fR, \fB--persistent\fR\fR
4148 .ad
4149 .sp .6
4150 .RS 4n
4151 Display the persistent IP tunnel configuration.
4152 .RE
```



```
4154 .sp
4155 .ne 2
4156 .na
4157 \fB\fB-p\fR, \fB--parseable\fR\fR
4158 .ad
4159 .sp .6
4160 .RS 4n
4161 Display using a stable machine-parseable format. The -o option is required with
4162 -p. See "Parseable Output Format", below.
4163 .RE

4165 .sp
4166 .ne 2
4167 .na
4168 \fB\fB-o\fR \fIfield\fR[,...], \fB--output\fR=\fIfield\fR[,...]\fR
4169 .ad
4170 .sp .6
4171 .RS 4n
4172 A case-insensitive, comma-separated list of output fields to display. The field
4173 name must be one of the fields listed below, or the special value \fBall\fR, to
4174 display all fields. By default (without \fB-o\fR), \fBshow-iptun\fR displays
4175 all fields.
4176 .sp
4177 .ne 2
4178 .na
4179 \fB\fBLINK\fR\fR
4180 .ad
4181 .sp .6
4182 .RS 4n
4183 The name of the IP tunnel link.
4184 .RE

4186 .sp
4187 .ne 2
4188 .na
4189 \fB\fBTYPE\fR\fR
4190 .ad
4191 .sp .6
4192 .RS 4n
4193 Type of tunnel as specified by the \fB-T\fR option of \fBcreate-iptun\fR.
4194 .RE

4196 .sp
4197 .ne 2
4198 .na
4199 \fB\fBFLAGS\fR\fR
4200 .ad
4201 .sp .6
4202 .RS 4n
4203 A set of flags associated with the IP tunnel link. Possible flags are:
4204 .sp
4205 .ne 2
4206 .na
4207 \fB\fBs\fR\fR
4208 .ad
4209 .sp .6
4210 .RS 4n
4211 The IP tunnel link is protected by IPsec policy. To display the IPsec policy
4212 associated with the tunnel link, enter:
4213 .sp
4214 .in +2
4215 .nf
4216 # \fBipsecconf -ln -i \fItunnel-link\fR\fR
4217 .fi
4218 .in -2
```

```
4219 .sp

4221 See \fBipsecconf\fR(1M) for more details on how to configure IPsec policy.
4222 .RE

4224 .sp
4225 .ne 2
4226 .na
4227 \fB\fBi\fR\fR
4228 .ad
4229 .sp .6
4230 .RS 4n
4231 The IP tunnel link was implicitly created with \fBifconfig\fR(1M), and will be
4232 automatically deleted when it is no longer referenced (that is, when the last
4233 IP interface over the tunnel is unplumbed). See \fBifconfig\fR(1M) for details
4234 on implicit tunnel creation.
4235 .RE

4237 .RE

4239 .sp
4240 .ne 2
4241 .na
4242 \fB\fBSOURCE\fR\fR
4243 .ad
4244 .sp .6
4245 .RS 4n
4246 The tunnel source address.
4247 .RE

4249 .sp
4250 .ne 2
4251 .na
4252 \fB\fBDESTINATION\fR\fR
4253 .ad
4254 .sp .6
4255 .RS 4n
4256 The tunnel destination address.
4257 .RE

4259 .RE

4261 .RE

4263 .sp
4264 .ne 2
4265 .na
4266 \fB\fBdladm show-usage\fR [\fB-a\fR] \fB-f\fR \fIfilename\fR [\fB-p\fR
4267 \fIplotfile\fR \fB-F\fR \fIformat\fR] [\fB-s\fR \fItime\fR] [\fB-e\fR
4268 \fItime\fR] [\fIlink\fR]\fR
4269 .ad
4270 .sp .6
4271 .RS 4n
4272 Show the historical network usage from a stored extended accounting file.
4273 Configuration and enabling of network accounting through \fBacctadm\fR(1M) is
4274 required. The default output will be the summary of network usage for the
4275 entire period of time in which extended accounting was enabled.
4276 .sp
4277 .ne 2
4278 .na
4279 \fB\fB-a\fR\fR
4280 .ad
4281 .sp .6
4282 .RS 4n
4283 Display all historical network usage for the specified period of time during
4284 which extended accounting is enabled. This includes the usage information for
```



```
4285 the links that have already been deleted.
4286 .RE

4288 .sp
4289 .ne 2
4290 .na
4291 \fB\fB-f\fR \fIfilename\fR, \fB--file\fR=\fIfilename\fR\fR
4292 .ad
4293 .sp .6
4294 .RS 4n
4295 Read extended accounting records of network usage from \fIfilename\fR.
4296 .RE

4298 .sp
4299 .ne 2
4300 .na
4301 \fB\fB-F\fR \fIformat\fR, \fB--format\fR=\fIformat\fR\fR
4302 .ad
4303 .sp .6
4304 .RS 4n
4305 Specifies the format of \fIplotfile\fR that is specified by the \fB-p\fR
4306 option. As of this release, \fBgnuplot\fR is the only supported format.
4307 .RE

4309 .sp
4310 .ne 2
4311 .na
4312 \fB\fB-p\fR \fIplotfile\fR, \fB--plot\fR=\fIplotfile\fR\fR
4313 .ad
4314 .sp .6
4315 .RS 4n
4316 Write network usage data to a file of the format specified by the \fB-F\fR
4317 option, which is required.
4318 .RE

4320 .sp
4321 .ne 2
4322 .na
4323 \fB\fB-s\fR \fItime\fR, \fB--start\fR=\fItime\fR\fR
4324 .ad
4325 .br
4326 .na
4327 \fB\fB-e\fR \fItime\fR, \fB--stop\fR=\fItime\fR\fR
4328 .ad
4329 .sp .6
4330 .RS 4n
4331 Start and stop times for data display. Time is in the format
4332 \fIMM\fR/\fIDD\fR/\fIYYYY\fR,\fIhh\fR:\fImm\fR:\fIss\fR.
4333 .RE

4335 .sp
4336 .ne 2
4337 .na
4338 \fB\fIlink\fR\fR
4339 .ad
4340 .sp .6
4341 .RS 4n
4342 If specified, display the network usage only for the named link. Otherwise,
4343 display network usage for all links.
4344 .RE

4346 .RE

4348 .SS "Parseable Output Format"
4349 .sp
4350 .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\fBmac-address\fR\fR
.ad
.sp .6
.RS 4n
The MAC address of the link. The default value is the factory MAC address.
.RE

.sp
.ne 2
.na
#endif /* ! codereview */
\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.
```

```
**********************************************************
  213717 Thu Feb 20 18:59:05 2014
new/usr/src/uts/common/io/mac/mac.c
2553 mac address should be a dladm link property
**********************************************************
______unchanged_portion_omitted_

2846 /*
2847  * Checks the size of the value size specified for a property as
2848  * part of a property operation. Returns B_TRUE if the size is
2849  * correct, B_FALSE otherwise.
2850  */
2851 boolean_t
2852 mac_prop_check_size(mac_prop_id_t id, uint_t valsize, boolean_t is_range)
2853 {
2854         uint_t minsize = 0;

2856         if (is_range)
2857                 return (valsize >= sizeof (mac_propval_range_t));

2859         switch (id) {
2860         case MAC_PROP_ZONE:
2861                 minsize = sizeof (dld_ioc_zid_t);
2862                 break;
2863         case MAC_PROP_AUTOPUSH:
2864                 if (valsize != 0)
2865                         minsize = sizeof (struct dlautopush);
2866                 break;
2867         case MAC_PROP_TAGMODE:
2868                 minsize = sizeof (link_tagmode_t);
2869                 break;
2870         case MAC_PROP_RESOURCE:
2871         case MAC_PROP_RESOURCE_EFF:
2872                 minsize = sizeof (mac_resource_props_t);
2873                 break;
2874         case MAC_PROP_DUPLEX:
2875                 minsize = sizeof (link_duplex_t);
2876                 break;
2877         case MAC_PROP_SPEED:
2878                 minsize = sizeof (uint64_t);
2879                 break;
2880         case MAC_PROP_STATUS:
2881                 minsize = sizeof (link_state_t);
2882                 break;
2883         case MAC_PROP_AUTONEG:
2884         case MAC_PROP_EN_AUTONEG:
2885                 minsize = sizeof (uint8_t);
2886                 break;
2887         case MAC_PROP_MTU:
2888         case MAC_PROP_LLIMIT:
2889         case MAC_PROP_LDECAY:
2890                 minsize = sizeof (uint32_t);
2891                 break;
2892         case MAC_PROP_FLOWCTRL:
2893                 minsize = sizeof (link_flowctrl_t);
2894                 break;
2895         case MAC_PROP_ADV_10GFDX_CAP:
2896         case MAC_PROP_EN_10GFDX_CAP:
2897         case MAC_PROP_ADV_1000HDX_CAP:
2898         case MAC_PROP_EN_1000HDX_CAP:
2899         case MAC_PROP_ADV_100FDX_CAP:
2900         case MAC_PROP_EN_100FDX_CAP:
2901         case MAC_PROP_ADV_100HDX_CAP:
2902         case MAC_PROP_EN_100HDX_CAP:
2903         case MAC_PROP_ADV_10FDX_CAP:
2904         case MAC_PROP_EN_10FDX_CAP:
```



```
2905         case MAC_PROP_ADV_10HDX_CAP:
2906         case MAC_PROP_EN_10HDX_CAP:
2907         case MAC_PROP_ADV_100T4_CAP:
2908         case MAC_PROP_EN_100T4_CAP:
2909                 minsize = sizeof (uint8_t);
2910                 break;
2911         case MAC_PROP_PVID:
2912                 minsize = sizeof (uint16_t);
2913                 break;
2914         case MAC_PROP_IPTUN_HOPLIMIT:
2915                 minsize = sizeof (uint32_t);
2916                 break;
2917         case MAC_PROP_IPTUN_ENCAPLIMIT:
2918                 minsize = sizeof (uint32_t);
2919                 break;
2920         case MAC_PROP_MAX_TX_RINGS_AVAIL:
2921         case MAC_PROP_MAX_RX_RINGS_AVAIL:
2922         case MAC_PROP_MAX_RXHWCLNT_AVAIL:
2923         case MAC_PROP_MAX_TXHWCLNT_AVAIL:
2924                 minsize = sizeof (uint_t);
2925                 break;
2926         case MAC_PROP_WL_ESSID:
2927                 minsize = sizeof (wl_linkstatus_t);
2928                 break;
2929         case MAC_PROP_WL_BSSID:
2930                 minsize = sizeof (wl_bssid_t);
2931                 break;
2932         case MAC_PROP_WL_BSSTYPE:
2933                 minsize = sizeof (wl_bss_type_t);
2934                 break;
2935         case MAC_PROP_WL_LINKSTATUS:
2936                 minsize = sizeof (wl_linkstatus_t);
2937                 break;
2938         case MAC_PROP_WL_DESIRED_RATES:
2939                 minsize = sizeof (wl_rates_t);
2940                 break;
2941         case MAC_PROP_WL_SUPPORTED_RATES:
2942                 minsize = sizeof (wl_rates_t);
2943                 break;
2944         case MAC_PROP_WL_AUTH_MODE:
2945                 minsize = sizeof (wl_authmode_t);
2946                 break;
2947         case MAC_PROP_WL_ENCRYPTION:
2948                 minsize = sizeof (wl_encryption_t);
2949                 break;
2950         case MAC_PROP_WL_RSSI:
2951                 minsize = sizeof (wl_rssi_t);
2952                 break;
2953         case MAC_PROP_WL_PHY_CONFIG:
2954                 minsize = sizeof (wl_phy_conf_t);
2955                 break;
2956         case MAC_PROP_WL_CAPABILITY:
2957                 minsize = sizeof (wl_capability_t);
2958                 break;
2959         case MAC_PROP_WL_WPA:
2960                 minsize = sizeof (wl_wpa_t);
2961                 break;
2962         case MAC_PROP_WL_SCANRESULTS:
2963                 minsize = sizeof (wl_wpa_ess_t);
2964                 break;
2965         case MAC_PROP_WL_POWER_MODE:
2966                 minsize = sizeof (wl_ps_mode_t);
2967                 break;
2968         case MAC_PROP_WL_RADIO:
2969                 minsize = sizeof (wl_radio_t);
2970                 break;
```

```
2971         case MAC_PROP_WL_ESS_LIST:
2972                 minsize = sizeof (wl_ess_list_t);
2973                 break;
2974         case MAC_PROP_WL_KEY_TAB:
2975                 minsize = sizeof (wl_wep_key_tab_t);
2976                 break;
2977         case MAC_PROP_WL_CREATE_IBSS:
2978                 minsize = sizeof (wl_create_ibss_t);
2979                 break;
2980         case MAC_PROP_WL_SETOPTIE:
2981                 minsize = sizeof (wl_wpa_ie_t);
2982                 break;
2983         case MAC_PROP_WL_DELKEY:
2984                 minsize = sizeof (wl_del_key_t);
2985                 break;
2986         case MAC_PROP_WL_KEY:
2987                 minsize = sizeof (wl_key_t);
2988                 break;
2989         case MAC_PROP_WL_MLME:
2990                 minsize = sizeof (wl_mlme_t);
2991                 break;
2992         case MAC_PROP_MACADDRESS:
2993                 minsize = sizeof (mac_addrprop_t);
2994 #endif /* ! codereview */
2995         }

2997         return (valsize >= minsize);
2998 }

3000 /*
3001  * mac_set_prop() sets MAC or hardware driver properties:
3002  *
3003  * - MAC-managed properties such as resource properties include maxbw,
3004  *   priority, and cpu binding list, as well as the default port VID
3005  *   used by bridging. These properties are consumed by the MAC layer
3006  *   itself and not passed down to the driver. For resource control
3007  *   properties, this function invokes mac_set_resources() which will
3008  *   cache the property value in mac_impl_t and may call
3009  *   mac_client_set_resource() to update property value of the primary
3010  *   mac client, if it exists.
3011  *
3012  * - Properties which act on the hardware and must be passed to the
3013  *   driver, such as MTU, through the driver's mc_setprop() entry point.
3014  */
3015 int
3016 mac_set_prop(mac_handle_t mh, mac_prop_id_t id, char *name, void *val,
3017     uint_t valsize)
3018 {
3019         int err = ENOTSUP;
3020         mac_impl_t *mip = (mac_impl_t *)mh;

3022         ASSERT(MAC_PERIM_HELD(mh));

3024         switch (id) {
3025         case MAC_PROP_RESOURCE: {
3026                 mac_resource_props_t *mrp;

3028                 /* call mac_set_resources() for MAC properties */
3029                 ASSERT(valsize >= sizeof (mac_resource_props_t));
3030                 mrp = kmem_zalloc(sizeof (*mrp), KM_SLEEP);
3031                 bcopy(val, mrp, sizeof (*mrp));
3032                 err = mac_set_resources(mh, mrp);
3033                 kmem_free(mrp, sizeof (*mrp));
3034                 break;
3035         }
```



```
3037         case MAC_PROP_PVID:
3038                 ASSERT(valsize >= sizeof (uint16_t));
3039                 if (mip->mi_state_flags & MIS_IS_VNIC)
3040                         return (EINVAL);
3041                 err = mac_set_pvid(mh, *(uint16_t *)val);
3042                 break;

3044         case MAC_PROP_MTU: {
3045                 uint32_t mtu;

3047                 ASSERT(valsize >= sizeof (uint32_t));
3048                 bcopy(val, &mtu, sizeof (mtu));
3049                 err = mac_set_mtu(mh, mtu, NULL);
3050                 break;
3051         }

3053         case MAC_PROP_LLIMIT:
3054         case MAC_PROP_LDECAY: {
3055                 uint32_t learnval;

3057                 if (valsize < sizeof (learnval) ||
3058                     (mip->mi_state_flags & MIS_IS_VNIC))
3059                         return (EINVAL);
3060                 bcopy(val, &learnval, sizeof (learnval));
3061                 if (learnval == 0 && id == MAC_PROP_LDECAY)
3062                         return (EINVAL);
3063                 if (id == MAC_PROP_LLIMIT)
3064                         mip->mi_llimit = learnval;
3065                 else
3066                         mip->mi_ldecay = learnval;
3067                 err = 0;
3068                 break;
3069         }

3071         case MAC_PROP_MACADDRESS: {
3072                 mac_addrprop_t  *addrprop = val;

3074                 if (addrprop->ma_len != mip->mi_type->mt_addr_length)
3075                         return (EINVAL);

3077                 err = mac_unicast_primary_set(mh, addrprop->ma_addr);
3078                 break;
3079         }

3081 #endif /* ! codereview */
3082         default:
3083                 /* For other driver properties, call driver's callback */
3084                 if (mip->mi_callbacks->mc_callbacks & MC_SETPROP) {
3085                         err = mip->mi_callbacks->mc_setprop(mip->mi_driver,
3086                             name, id, valsize, val);
3087                 }
3088         }
3089         return (err);
3090 }

3092 /*
3093  * mac_get_prop() gets MAC or device driver properties.
3094  *
3095  * If the property is a driver property, mac_get_prop() calls driver's callback
3096  * entry point to get it.
3097  * If the property is a MAC property, mac_get_prop() invokes mac_get_resources()
3098  * which returns the cached value in mac_impl_t.
3099  */
3100 int
3101 mac_get_prop(mac_handle_t mh, mac_prop_id_t id, char *name, void *val,
3102     uint_t valsize)
```

```
3103 {
3104         int err = ENOTSUP;
3105         mac_impl_t *mip = (mac_impl_t *)mh;
3106         uint_t  rings;
3107         uint_t  vlinks;
3109         bzero(val, valsize);
3111         switch (id) {
3112         case MAC_PROP_RESOURCE: {
3113                 mac_resource_props_t *mrp;
3115                 /* If mac property, read from cache */
3116                 ASSERT(valsize >= sizeof (mac_resource_props_t));
3117                 mrp = kmem_zalloc(sizeof (*mrp), KM_SLEEP);
3118                 mac_get_resources(mh, mrp);
3119                 bcopy(mrp, val, sizeof (*mrp));
3120                 kmem_free(mrp, sizeof (*mrp));
3121                 return (0);
3122         }
3123         case MAC_PROP_RESOURCE_EFF: {
3124                 mac_resource_props_t *mrp;
3126                 /* If mac effective property, read from client */
3127                 ASSERT(valsize >= sizeof (mac_resource_props_t));
3128                 mrp = kmem_zalloc(sizeof (*mrp), KM_SLEEP);
3129                 mac_get_effective_resources(mh, mrp);
3130                 bcopy(mrp, val, sizeof (*mrp));
3131                 kmem_free(mrp, sizeof (*mrp));
3132                 return (0);
3133         }
3135         case MAC_PROP_PVID:
3136                 ASSERT(valsize >= sizeof (uint16_t));
3137                 if (mip->mi_state_flags & MIS_IS_VNIC)
3138                         return (EINVAL);
3139                 *(uint16_t *)val = mac_get_pvid(mh);
3140                 return (0);
3142         case MAC_PROP_LLIMIT:
3143         case MAC_PROP_LDECAY:
3144                 ASSERT(valsize >= sizeof (uint32_t));
3145                 if (mip->mi_state_flags & MIS_IS_VNIC)
3146                         return (EINVAL);
3147                 if (id == MAC_PROP_LLIMIT)
3148                         bcopy(&mip->mi_llimit, val, sizeof (mip->mi_llimit));
3149                 else
3150                         bcopy(&mip->mi_ldecay, val, sizeof (mip->mi_ldecay));
3151                 return (0);
3153         case MAC_PROP_MTU: {
3154                 uint32_t sdu;
3156                 ASSERT(valsize >= sizeof (uint32_t));
3157                 mac_sdu_get2(mh, NULL, &sdu, NULL);
3158                 bcopy(&sdu, val, sizeof (sdu));
3160                 return (0);
3161         }
3162         case MAC_PROP_STATUS: {
3163                 link_state_t link_state;
3165                 if (valsize < sizeof (link_state))
3166                         return (EINVAL);
3167                 link_state = mac_link_get(mh);
3168                 bcopy(&link_state, val, sizeof (link_state));
```



```
3170                 return (0);
3171         }
3173         case MAC_PROP_MAX_RX_RINGS_AVAIL:
3174         case MAC_PROP_MAX_TX_RINGS_AVAIL:
3175                 ASSERT(valsize >= sizeof (uint_t));
3176                 rings = id == MAC_PROP_MAX_RX_RINGS_AVAIL ?
3177                     mac_rxavail_get(mh) : mac_txavail_get(mh);
3178                 bcopy(&rings, val, sizeof (uint_t));
3179                 return (0);
3181         case MAC_PROP_MAX_RXHWCLNT_AVAIL:
3182         case MAC_PROP_MAX_TXHWCLNT_AVAIL:
3183                 ASSERT(valsize >= sizeof (uint_t));
3184                 vlinks = id == MAC_PROP_MAX_RXHWCLNT_AVAIL ?
3185                     mac_rxhwlnksavail_get(mh) : mac_txhwlnksavail_get(mh);
3186                 bcopy(&vlinks, val, sizeof (uint_t));
3187                 return (0);
3189         case MAC_PROP_RXRINGSRANGE:
3190         case MAC_PROP_TXRINGSRANGE:
3191                 /*
3192                  * The value for these properties are returned through
3193                  * the MAC_PROP_RESOURCE property.
3194                  */
3195                 return (0);
3197         case MAC_PROP_MACADDRESS: {
3198                 mac_addrprop_t  *addrprop = val;
3200                 if (valsize < sizeof (mac_addrprop_t))
3201                         return (EINVAL);
3202                 mac_unicast_primary_get(mh, addrprop->ma_addr);
3203                 addrprop->ma_len = mip->mi_type->mt_addr_length;
3204                 return (0);
3205         }
3207 #endif /* ! codereview */
3208         default:
3209                 break;
3211         }
3213         /* If driver property, request from driver */
3214         if (mip->mi_callbacks->mc_callbacks & MC_GETPROP) {
3215                 err = mip->mi_callbacks->mc_getprop(mip->mi_driver, name, id,
3216                     valsize, val);
3217         }
3219         return (err);
3220 }
3222 /*
3223  * Helper function to initialize the range structure for use in
3224  * mac_get_prop. If the type can be other than uint32, we can
3225  * pass that as an arg.
3226  */
3227 static void
3228 _mac_set_range(mac_propval_range_t *range, uint32_t min, uint32_t max)
3229 {
3230         range->mpr_count = 1;
3231         range->mpr_type = MAC_PROPVAL_UINT32;
3232         range->mpr_range_uint32[0].mpur_min = min;
3233         range->mpr_range_uint32[0].mpur_max = max;
3234 }
```

```
3236 /*
3237  * Returns information about the specified property, such as default
3238  * values or permissions.
3239  */
3240 int
3241 mac_prop_info(mac_handle_t mh, mac_prop_id_t id, char *name,
3242     void *default_val, uint_t default_size, mac_propval_range_t *range,
3243     uint_t *perm)
3244 {
3245         mac_prop_info_state_t state;
3246         mac_impl_t *mip = (mac_impl_t *)mh;
3247         uint_t  max;

3249         /*
3250          * A property is read/write by default unless the driver says
3251          * otherwise.
3252          */
3253         if (perm != NULL)
3254                 *perm = MAC_PROP_PERM_RW;

3256         if (default_val != NULL)
3257                 bzero(default_val, default_size);

3259         /*
3260          * First, handle framework properties for which we don't need to
3261          * involve the driver.
3262          */
3263         switch (id) {
3264         case MAC_PROP_RESOURCE:
3265         case MAC_PROP_PVID:
3266         case MAC_PROP_LLIMIT:
3267         case MAC_PROP_LDECAY:
3268                 return (0);

3270         case MAC_PROP_MAX_RX_RINGS_AVAIL:
3271         case MAC_PROP_MAX_TX_RINGS_AVAIL:
3272         case MAC_PROP_MAX_RXHWCLNT_AVAIL:
3273         case MAC_PROP_MAX_TXHWCLNT_AVAIL:
3274                 if (perm != NULL)
3275                         *perm = MAC_PROP_PERM_READ;
3276                 return (0);

3278         case MAC_PROP_RXRINGSRANGE:
3279         case MAC_PROP_TXRINGSRANGE:
3280                 /*
3281                  * Currently, we support range for RX and TX rings properties.
3282                  * When we extend this support to maxbw, cpus and priority,
3283                  * we should move this to mac_get_resources.
3284                  * There is no default value for RX or TX rings.
3285                  */
3286                 if ((mip->mi_state_flags & MIS_IS_VNIC) &&
3287                     mac_is_vnic_primary(mh)) {
3288                         /*
3289                          * We don't support setting rings for a VLAN
3290                          * data link because it shares its ring with the
3291                          * primary MAC client.
3292                          */
3293                         if (perm != NULL)
3294                                 *perm = MAC_PROP_PERM_READ;
3295                         if (range != NULL)
3296                                 range->mpr_count = 0;
3297                 } else if (range != NULL) {
3298                         if (mip->mi_state_flags & MIS_IS_VNIC)
3299                                 mh = mac_get_lower_mac_handle(mh);
3300                         mip = (mac_impl_t *)mh;
```



```
3301                         if ((id == MAC_PROP_RXRINGSRANGE &&
3302                             mip->mi_rx_group_type == MAC_GROUP_TYPE_STATIC) ||
3303                             (id == MAC_PROP_TXRINGSRANGE &&
3304                             mip->mi_tx_group_type == MAC_GROUP_TYPE_STATIC)) {
3305                                 if (id == MAC_PROP_RXRINGSRANGE) {
3306                                         if ((mac_rxhwlnksavail_get(mh) +
3307                                             mac_rxhwlnksrsvd_get(mh)) <= 1) {
3308                                                 /*
3309                                                  * doesn't support groups or
3310                                                  * rings
3311                                                  */
3312                                                 range->mpr_count = 0;
3313                                         } else {
3314                                                 /*
3315                                                  * supports specifying groups,
3316                                                  * but not rings
3317                                                  */
3318                                                 _mac_set_range(range, 0, 0);
3319                                         }
3320                                 } else {
3321                                         if ((mac_txhwlnksavail_get(mh) +
3322                                             mac_txhwlnksrsvd_get(mh)) <= 1) {
3323                                                 /*
3324                                                  * doesn't support groups or
3325                                                  * rings
3326                                                  */
3327                                                 range->mpr_count = 0;
3328                                         } else {
3329                                                 /*
3330                                                  * supports specifying groups,
3331                                                  * but not rings
3332                                                  */
3333                                                 _mac_set_range(range, 0, 0);
3334                                         }
3335                                 }
3336                         } else {
3337                                 max = id == MAC_PROP_RXRINGSRANGE ?
3338                                     mac_rxavail_get(mh) + mac_rxrsvd_get(mh) :
3339                                     mac_txavail_get(mh) + mac_txrsvd_get(mh);
3340                                 if (max <= 1) {
3341                                         /*
3342                                          * doesn't support groups or
3343                                          * rings
3344                                          */
3345                                         range->mpr_count = 0;
3346                                 } else  {
3347                                         /*
3348                                          * -1 because we have to leave out the
3349                                          * default ring.
3350                                          */
3351                                         _mac_set_range(range, 1, max - 1);
3352                                 }
3353                         }
3354                 }
3355                 return (0);

3357         case MAC_PROP_STATUS:
3358                 if (perm != NULL)
3359                         *perm = MAC_PROP_PERM_READ;
3360                 return (0);

3362         case MAC_PROP_MACADDRESS: {
3363                 mac_addrprop_t  *defaddr = default_val;

3365                 if (defaddr != NULL) {
3366                         if (default_size < sizeof (mac_addrprop_t))
```

```
3367                                return (EINVAL);
3368                        bcopy(mip->mi_info.mi_unicst_addr, defaddr->ma_addr,
3369                            mip->mi_type->mt_addr_length);
3370                        defaddr->ma_len = mip->mi_type->mt_addr_length;
3371                }
3372                return (0);
3373        }
3374 #endif /* ! codereview */
3375        }

3377        /*
3378         * Get the property info from the driver if it implements the
3379         * property info entry point.
3380         */
3381        bzero(&state, sizeof (state));

3383        if (mip->mi_callbacks->mc_callbacks & MC_PROPINFO) {
3384                state.pr_default = default_val;
3385                state.pr_default_size = default_size;

3387                /*
3388                 * The caller specifies the maximum number of ranges
3389                 * it can accomodate using mpr_count. We don't touch
3390                 * this value until the driver returns from its
3391                 * mc_propinfo() callback, and ensure we don't exceed
3392                 * this number of range as the driver defines
3393                 * supported range from its mc_propinfo().
3394                 *
3395                 * pr_range_cur_count keeps track of how many ranges
3396                 * were defined by the driver from its mc_propinfo()
3397                 * entry point.
3398                 *
3399                 * On exit, the user-specified range mpr_count returns
3400                 * the number of ranges specified by the driver on
3401                 * success, or the number of ranges it wanted to
3402                 * define if that number of ranges could not be
3403                 * accomodated by the specified range structure.  In
3404                 * the latter case, the caller will be able to
3405                 * allocate a larger range structure, and query the
3406                 * property again.
3407                 */
3408                state.pr_range_cur_count = 0;
3409                state.pr_range = range;

3411                mip->mi_callbacks->mc_propinfo(mip->mi_driver, name, id,
3412                    (mac_prop_info_handle_t)&state);

3414                if (state.pr_flags & MAC_PROP_INFO_RANGE)
3415                        range->mpr_count = state.pr_range_cur_count;

3417                /*
3418                 * The operation could fail if the buffer supplied by
3419                 * the user was too small for the range or default
3420                 * value of the property.
3421                 */
3422                if (state.pr_errno != 0)
3423                        return (state.pr_errno);

3425                if (perm != NULL && state.pr_flags & MAC_PROP_INFO_PERM)
3426                        *perm = state.pr_perm;
3427        }

3429        /*
3430         * The MAC layer may want to provide default values or allowed
3431         * ranges for properties if the driver does not provide a
3432         * property info entry point, or that entry point exists, but
```



```
3433         * it did not provide a default value or allowed ranges for
3434         * that property.
3435         */
3436        switch (id) {
3437        case MAC_PROP_MTU: {
3438                uint32_t sdu;

3440                mac_sdu_get2(mh, NULL, &sdu, NULL);

3442                if (range != NULL && !(state.pr_flags &
3443                    MAC_PROP_INFO_RANGE)) {
3444                        /* MTU range */
3445                        _mac_set_range(range, sdu, sdu);
3446                }

3448                if (default_val != NULL && !(state.pr_flags &
3449                    MAC_PROP_INFO_DEFAULT)) {
3450                        if (mip->mi_info.mi_media == DL_ETHER)
3451                                sdu = ETHERMTU;
3452                        /* default MTU value */
3453                        bcopy(&sdu, default_val, sizeof (sdu));
3454                }
3455        }
3456        }

3458        return (0);
3459 }

3461 int
3462 mac_fastpath_disable(mac_handle_t mh)
3463 {
3464        mac_impl_t      *mip = (mac_impl_t *)mh;

3466        if ((mip->mi_state_flags & MIS_LEGACY) == 0)
3467                return (0);

3469        return (mip->mi_capab_legacy.ml_fastpath_disable(mip->mi_driver));
3470 }

3472 void
3473 mac_fastpath_enable(mac_handle_t mh)
3474 {
3475        mac_impl_t      *mip = (mac_impl_t *)mh;

3477        if ((mip->mi_state_flags & MIS_LEGACY) == 0)
3478                return;

3480        mip->mi_capab_legacy.ml_fastpath_enable(mip->mi_driver);
3481 }

3483 void
3484 mac_register_priv_prop(mac_impl_t *mip, char **priv_props)
3485 {
3486        uint_t nprops, i;

3488        if (priv_props == NULL)
3489                return;

3491        nprops = 0;
3492        while (priv_props[nprops] != NULL)
3493                nprops++;
3494        if (nprops == 0)
3495                return;


3498        mip->mi_priv_prop = kmem_zalloc(nprops * sizeof (char *), KM_SLEEP);
```

```
3500         for (i = 0; i < nprops; i++) {
3501                 mip->mi_priv_prop[i] = kmem_zalloc(MAXLINKPROPNAME, KM_SLEEP);
3502                 (void) strlcpy(mip->mi_priv_prop[i], priv_props[i],
3503                     MAXLINKPROPNAME);
3504         }

3506         mip->mi_priv_prop_count = nprops;
3507 }

3509 void
3510 mac_unregister_priv_prop(mac_impl_t *mip)
3511 {
3512         uint_t i;

3514         if (mip->mi_priv_prop_count == 0) {
3515                 ASSERT(mip->mi_priv_prop == NULL);
3516                 return;
3517         }

3519         for (i = 0; i < mip->mi_priv_prop_count; i++)
3520                 kmem_free(mip->mi_priv_prop[i], MAXLINKPROPNAME);
3521         kmem_free(mip->mi_priv_prop, mip->mi_priv_prop_count *
3522             sizeof (char *));

3524         mip->mi_priv_prop = NULL;
3525         mip->mi_priv_prop_count = 0;
3526 }

3528 /*
3529  * mac_ring_t 'mr' macros. Some rogue drivers may access ring structure
3530  * (by invoking mac_rx()) even after processing mac_stop_ring(). In such
3531  * cases if MAC free's the ring structure after mac_stop_ring(), any
3532  * illegal access to the ring structure coming from the driver will panic
3533  * the system. In order to protect the system from such inadverent access,
3534  * we maintain a cache of rings in the mac_impl_t after they get free'd up.
3535  * When packets are received on free'd up rings, MAC (through the generation
3536  * count mechanism) will drop such packets.
3537  */
3538 static mac_ring_t *
3539 mac_ring_alloc(mac_impl_t *mip)
3540 {
3541         mac_ring_t *ring;

3543         mutex_enter(&mip->mi_ring_lock);
3544         if (mip->mi_ring_freelist != NULL) {
3545                 ring = mip->mi_ring_freelist;
3546                 mip->mi_ring_freelist = ring->mr_next;
3547                 bzero(ring, sizeof (mac_ring_t));
3548                 mutex_exit(&mip->mi_ring_lock);
3549         } else {
3550                 mutex_exit(&mip->mi_ring_lock);
3551                 ring = kmem_cache_alloc(mac_ring_cache, KM_SLEEP);
3552         }
3553         ASSERT((ring != NULL) && (ring->mr_state == MR_FREE));
3554         return (ring);
3555 }

3557 static void
3558 mac_ring_free(mac_impl_t *mip, mac_ring_t *ring)
3559 {
3560         ASSERT(ring->mr_state == MR_FREE);

3562         mutex_enter(&mip->mi_ring_lock);
3563         ring->mr_state = MR_FREE;
3564         ring->mr_flag = 0;
```



```
3565         ring->mr_next = mip->mi_ring_freelist;
3566         ring->mr_mip = NULL;
3567         mip->mi_ring_freelist = ring;
3568         mac_ring_stat_delete(ring);
3569         mutex_exit(&mip->mi_ring_lock);
3570 }

3572 static void
3573 mac_ring_freeall(mac_impl_t *mip)
3574 {
3575         mac_ring_t *ring_next;
3576         mutex_enter(&mip->mi_ring_lock);
3577         mac_ring_t *ring = mip->mi_ring_freelist;
3578         while (ring != NULL) {
3579                 ring_next = ring->mr_next;
3580                 kmem_cache_free(mac_ring_cache, ring);
3581                 ring = ring_next;
3582         }
3583         mip->mi_ring_freelist = NULL;
3584         mutex_exit(&mip->mi_ring_lock);
3585 }

3587 int
3588 mac_start_ring(mac_ring_t *ring)
3589 {
3590         int rv = 0;

3592         ASSERT(ring->mr_state == MR_FREE);

3594         if (ring->mr_start != NULL) {
3595                 rv = ring->mr_start(ring->mr_driver, ring->mr_gen_num);
3596                 if (rv != 0)
3597                         return (rv);
3598         }

3600         ring->mr_state = MR_INUSE;
3601         return (rv);
3602 }

3604 void
3605 mac_stop_ring(mac_ring_t *ring)
3606 {
3607         ASSERT(ring->mr_state == MR_INUSE);

3609         if (ring->mr_stop != NULL)
3610                 ring->mr_stop(ring->mr_driver);

3612         ring->mr_state = MR_FREE;

3614         /*
3615          * Increment the ring generation number for this ring.
3616          */
3617         ring->mr_gen_num++;
3618 }

3620 int
3621 mac_start_group(mac_group_t *group)
3622 {
3623         int rv = 0;

3625         if (group->mrg_start != NULL)
3626                 rv = group->mrg_start(group->mrg_driver);

3628         return (rv);
3629 }
```

```
3631 void
3632 mac_stop_group(mac_group_t *group)
3633 {
3634         if (group->mrg_stop != NULL)
3635                 group->mrg_stop(group->mrg_driver);
3636 }

3638 /*
3639  * Called from mac_start() on the default Rx group. Broadcast and multicast
3640  * packets are received only on the default group. Hence the default group
3641  * needs to be up even if the primary client is not up, for the other groups
3642  * to be functional. We do this by calling this function at mac_start time
3643  * itself. However the broadcast packets that are received can't make their
3644  * way beyond mac_rx until a mac client creates a broadcast flow.
3645  */
3646 static int
3647 mac_start_group_and_rings(mac_group_t *group)
3648 {
3649         mac_ring_t      *ring;
3650         int             rv = 0;

3652         ASSERT(group->mrg_state == MAC_GROUP_STATE_REGISTERED);
3653         if ((rv = mac_start_group(group)) != 0)
3654                 return (rv);

3656         for (ring = group->mrg_rings; ring != NULL; ring = ring->mr_next) {
3657                 ASSERT(ring->mr_state == MR_FREE);
3658                 if ((rv = mac_start_ring(ring)) != 0)
3659                         goto error;
3660                 ring->mr_classify_type = MAC_SW_CLASSIFIER;
3661         }
3662         return (0);

3664 error:
3665         mac_stop_group_and_rings(group);
3666         return (rv);
3667 }

3669 /* Called from mac_stop on the default Rx group */
3670 static void
3671 mac_stop_group_and_rings(mac_group_t *group)
3672 {
3673         mac_ring_t      *ring;

3675         for (ring = group->mrg_rings; ring != NULL; ring = ring->mr_next) {
3676                 if (ring->mr_state != MR_FREE) {
3677                         mac_stop_ring(ring);
3678                         ring->mr_flag = 0;
3679                         ring->mr_classify_type = MAC_NO_CLASSIFIER;
3680                 }
3681         }
3682         mac_stop_group(group);
3683 }


3686 static mac_ring_t *
3687 mac_init_ring(mac_impl_t *mip, mac_group_t *group, int index,
3688     mac_capab_rings_t *cap_rings)
3689 {
3690         mac_ring_t *ring, *rnext;
3691         mac_ring_info_t ring_info;
3692         ddi_intr_handle_t ddi_handle;

3694         ring = mac_ring_alloc(mip);

3696         /* Prepare basic information of ring */
```



```
3698         /*
3699          * Ring index is numbered to be unique across a particular device.
3700          * Ring index computation makes following assumptions:
3701          *      - For drivers with static grouping (e.g. ixgbe, bge),
3702          *      ring index exchanged with the driver (e.g. during mr_rget)
3703          *      is unique only across the group the ring belongs to.
3704          *      - Drivers with dynamic grouping (e.g. nxge), start
3705          *      with single group (mrg_index = 0).
3706          */
3707         ring->mr_index = group->mrg_index * group->mrg_info.mgi_count + index;
3708         ring->mr_type = group->mrg_type;
3709         ring->mr_gh = (mac_group_handle_t)group;

3711         /* Insert the new ring to the list. */
3712         ring->mr_next = group->mrg_rings;
3713         group->mrg_rings = ring;

3715         /* Zero to reuse the info data structure */
3716         bzero(&ring_info, sizeof (ring_info));

3718         /* Query ring information from driver */
3719         cap_rings->mr_rget(mip->mi_driver, group->mrg_type, group->mrg_index,
3720             index, &ring_info, (mac_ring_handle_t)ring);

3722         ring->mr_info = ring_info;

3724         /*
3725          * The interrupt handle could be shared among multiple rings.
3726          * Thus if there is a bunch of rings that are sharing an
3727          * interrupt, then only one ring among the bunch will be made
3728          * available for interrupt re-targeting; the rest will have
3729          * ddi_shared flag set to TRUE and would not be available for
3730          * be interrupt re-targeting.
3731          */
3732         if ((ddi_handle = ring_info.mri_intr.mi_ddi_handle) != NULL) {
3733                 rnext = ring->mr_next;
3734                 while (rnext != NULL) {
3735                         if (rnext->mr_info.mri_intr.mi_ddi_handle ==
3736                             ddi_handle) {
3737                                 /*
3738                                  * If default ring (mr_index == 0) is part
3739                                  * of a group of rings sharing an
3740                                  * interrupt, then set ddi_shared flag for
3741                                  * the default ring and give another ring
3742                                  * the chance to be re-targeted.
3743                                  */
3744                                 if (rnext->mr_index == 0 &&
3745                                     !rnext->mr_info.mri_intr.mi_ddi_shared) {
3746                                         rnext->mr_info.mri_intr.mi_ddi_shared =
3747                                             B_TRUE;
3748                                 } else {
3749                                         ring->mr_info.mri_intr.mi_ddi_shared =
3750                                             B_TRUE;
3751                                 }
3752                                 break;
3753                         }
3754                         rnext = rnext->mr_next;
3755                 }
3756                 /*
3757                  * If rnext is NULL, then no matching ddi_handle was found.
3758                  * Rx rings get registered first. So if this is a Tx ring,
3759                  * then go through all the Rx rings and see if there is a
3760                  * matching ddi handle.
3761                  */
3762                 if (rnext == NULL && ring->mr_type == MAC_RING_TYPE_TX) {
```

```
3763                        mac_compare_ddi_handle(mip->mi_rx_groups,
3764                            mip->mi_rx_group_count, ring);
3765                }
3766        }

3768        /* Update ring's status */
3769        ring->mr_state = MR_FREE;
3770        ring->mr_flag = 0;

3772        /* Update the ring count of the group */
3773        group->mrg_cur_count++;

3775        /* Create per ring kstats */
3776        if (ring->mr_stat != NULL) {
3777                ring->mr_mip = mip;
3778                mac_ring_stat_create(ring);
3779        }

3781        return (ring);
3782 }

3784 /*
3785  * Rings are chained together for easy regrouping.
3786  */
3787 static void
3788 mac_init_group(mac_impl_t *mip, mac_group_t *group, int size,
3789     mac_capab_rings_t *cap_rings)
3790 {
3791        int index;

3793        /*
3794         * Initialize all ring members of this group. Size of zero will not
3795         * enter the loop, so it's safe for initializing an empty group.
3796         */
3797        for (index = size - 1; index >= 0; index--)
3798                (void) mac_init_ring(mip, group, index, cap_rings);
3799 }

3801 int
3802 mac_init_rings(mac_impl_t *mip, mac_ring_type_t rtype)
3803 {
3804        mac_capab_rings_t       *cap_rings;
3805        mac_group_t             *group;
3806        mac_group_t             *groups;
3807        mac_group_info_t        group_info;
3808        uint_t                  group_free = 0;
3809        uint_t                  ring_left;
3810        mac_ring_t              *ring;
3811        int                     g;
3812        int                     err = 0;
3813        uint_t                  grpcnt;
3814        boolean_t               pseudo_txgrp = B_FALSE;

3816        switch (rtype) {
3817        case MAC_RING_TYPE_RX:
3818                ASSERT(mip->mi_rx_groups == NULL);

3820                cap_rings = &mip->mi_rx_rings_cap;
3821                cap_rings->mr_type = MAC_RING_TYPE_RX;
3822                break;
3823        case MAC_RING_TYPE_TX:
3824                ASSERT(mip->mi_tx_groups == NULL);

3826                cap_rings = &mip->mi_tx_rings_cap;
3827                cap_rings->mr_type = MAC_RING_TYPE_TX;
3828                break;
```



```
3829        default:
3830                ASSERT(B_FALSE);
3831        }

3833        if (!i_mac_capab_get((mac_handle_t)mip, MAC_CAPAB_RINGS, cap_rings))
3834                return (0);
3835        grpcnt = cap_rings->mr_gnum;

3837        /*
3838         * If we have multiple TX rings, but only one TX group, we can
3839         * create pseudo TX groups (one per TX ring) in the MAC layer,
3840         * except for an aggr. For an aggr currently we maintain only
3841         * one group with all the rings (for all its ports), going
3842         * forwards we might change this.
3843         */
3844        if (rtype == MAC_RING_TYPE_TX &&
3845            cap_rings->mr_gnum == 0 && cap_rings->mr_rnum >  0 &&
3846            (mip->mi_state_flags & MIS_IS_AGGR) == 0) {
3847                /*
3848                 * The -1 here is because we create a default TX group
3849                 * with all the rings in it.
3850                 */
3851                grpcnt = cap_rings->mr_rnum - 1;
3852                pseudo_txgrp = B_TRUE;
3853        }

3855        /*
3856         * Allocate a contiguous buffer for all groups.
3857         */
3858        groups = kmem_zalloc(sizeof (mac_group_t) * (grpcnt+ 1), KM_SLEEP);

3860        ring_left = cap_rings->mr_rnum;

3862        /*
3863         * Get all ring groups if any, and get their ring members
3864         * if any.
3865         */
3866        for (g = 0; g < grpcnt; g++) {
3867                group = groups + g;

3869                /* Prepare basic information of the group */
3870                group->mrg_index = g;
3871                group->mrg_type = rtype;
3872                group->mrg_state = MAC_GROUP_STATE_UNINIT;
3873                group->mrg_mh = (mac_handle_t)mip;
3874                group->mrg_next = group + 1;

3876                /* Zero to reuse the info data structure */
3877                bzero(&group_info, sizeof (group_info));

3879                if (pseudo_txgrp) {
3880                        /*
3881                         * This is a pseudo group that we created, apart
3882                         * from setting the state there is nothing to be
3883                         * done.
3884                         */
3885                        group->mrg_state = MAC_GROUP_STATE_REGISTERED;
3886                        group_free++;
3887                        continue;
3888                }
3889                /* Query group information from driver */
3890                cap_rings->mr_gget(mip->mi_driver, rtype, g, &group_info,
3891                    (mac_group_handle_t)group);

3893                switch (cap_rings->mr_group_type) {
3894                case MAC_GROUP_TYPE_DYNAMIC:
```

```
3895                        if (cap_rings->mr_gaddring == NULL ||
3896                            cap_rings->mr_gremring == NULL) {
3897                                DTRACE_PROBE3(
3898                                    mac__init__rings_no_addremring,
3899                                    char *, mip->mi_name,
3900                                    mac_group_add_ring_t,
3901                                    cap_rings->mr_gaddring,
3902                                    mac_group_add_ring_t,
3903                                    cap_rings->mr_gremring);
3904                                err = EINVAL;
3905                                goto bail;
3906                        }

3908                        switch (rtype) {
3909                        case MAC_RING_TYPE_RX:
3910                                /*
3911                                 * The first RX group must have non-zero
3912                                 * rings, and the following groups must
3913                                 * have zero rings.
3914                                 */
3915                                if (g == 0 && group_info.mgi_count == 0) {
3916                                        DTRACE_PROBE1(
3917                                            mac__init__rings__rx__def__zero,
3918                                            char *, mip->mi_name);
3919                                        err = EINVAL;
3920                                        goto bail;
3921                                }
3922                                if (g > 0 && group_info.mgi_count != 0) {
3923                                        DTRACE_PROBE3(
3924                                            mac__init__rings__rx__nonzero,
3925                                            char *, mip->mi_name,
3926                                            int, g, int, group_info.mgi_count);
3927                                        err = EINVAL;
3928                                        goto bail;
3929                                }
3930                                break;
3931                        case MAC_RING_TYPE_TX:
3932                                /*
3933                                 * All TX ring groups must have zero rings.
3934                                 */
3935                                if (group_info.mgi_count != 0) {
3936                                        DTRACE_PROBE3(
3937                                            mac__init__rings__tx__nonzero,
3938                                            char *, mip->mi_name,
3939                                            int, g, int, group_info.mgi_count);
3940                                        err = EINVAL;
3941                                        goto bail;
3942                                }
3943                                break;
3944                        }
3945                        break;
3946                case MAC_GROUP_TYPE_STATIC:
3947                        /*
3948                         * Note that an empty group is allowed, e.g., an aggr
3949                         * would start with an empty group.
3950                         */
3951                        break;
3952                default:
3953                        /* unknown group type */
3954                        DTRACE_PROBE2(mac__init__rings__unknown__type,
3955                            char *, mip->mi_name,
3956                            int, cap_rings->mr_group_type);
3957                        err = EINVAL;
3958                        goto bail;
3959                }
```



```
3962                /*
3963                 * Driver must register group->mgi_addmac/remmac() for rx groups
3964                 * to support multiple MAC addresses.
3965                 */
3966                if (rtype == MAC_RING_TYPE_RX) {
3967                        if ((group_info.mgi_addmac == NULL) ||
3968                            (group_info.mgi_addmac == NULL)) {
3969                                goto bail;
3970                        }
3971                }

3973                /* Cache driver-supplied information */
3974                group->mrg_info = group_info;

3976                /* Update the group's status and group count. */
3977                mac_set_group_state(group, MAC_GROUP_STATE_REGISTERED);
3978                group_free++;

3980                group->mrg_rings = NULL;
3981                group->mrg_cur_count = 0;
3982                mac_init_group(mip, group, group_info.mgi_count, cap_rings);
3983                ring_left -= group_info.mgi_count;

3985                /* The current group size should be equal to default value */
3986                ASSERT(group->mrg_cur_count == group_info.mgi_count);
3987        }

3989        /* Build up a dummy group for free resources as a pool */
3990        group = groups + grpcnt;

3992        /* Prepare basic information of the group */
3993        group->mrg_index = -1;
3994        group->mrg_type = rtype;
3995        group->mrg_state = MAC_GROUP_STATE_UNINIT;
3996        group->mrg_mh = (mac_handle_t)mip;
3997        group->mrg_next = NULL;

3999        /*
4000         * If there are ungrouped rings, allocate a continuous buffer for
4001         * remaining resources.
4002         */
4003        if (ring_left != 0) {
4004                group->mrg_rings = NULL;
4005                group->mrg_cur_count = 0;
4006                mac_init_group(mip, group, ring_left, cap_rings);

4008                /* The current group size should be equal to ring_left */
4009                ASSERT(group->mrg_cur_count == ring_left);

4011                ring_left = 0;

4013                /* Update this group's status */
4014                mac_set_group_state(group, MAC_GROUP_STATE_REGISTERED);
4015        } else
4016                group->mrg_rings = NULL;

4018        ASSERT(ring_left == 0);

4020 bail:

4022        /* Cache other important information to finalize the initialization */
4023        switch (rtype) {
4024        case MAC_RING_TYPE_RX:
4025                mip->mi_rx_group_type = cap_rings->mr_group_type;
4026                mip->mi_rx_group_count = cap_rings->mr_gnum;
```

```
4027                mip->mi_rx_groups = groups;
4028                mip->mi_rx_donor_grp = groups;
4029                if (mip->mi_rx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
4030                        /*
4031                         * The default ring is reserved since it is
4032                         * used for sending the broadcast etc. packets.
4033                         */
4034                        mip->mi_rxrings_avail =
4035                            mip->mi_rx_groups->mrg_cur_count - 1;
4036                        mip->mi_rxrings_rsvd = 1;
4037                }
4038                /*
4039                 * The default group cannot be reserved. It is used by
4040                 * all the clients that do not have an exclusive group.
4041                 */
4042                mip->mi_rxhwclnt_avail = mip->mi_rx_group_count - 1;
4043                mip->mi_rxhwclnt_used = 1;
4044                break;
4045        case MAC_RING_TYPE_TX:
4046                mip->mi_tx_group_type = pseudo_txgrp ? MAC_GROUP_TYPE_DYNAMIC :
4047                    cap_rings->mr_group_type;
4048                mip->mi_tx_group_count = grpcnt;
4049                mip->mi_tx_group_free = group_free;
4050                mip->mi_tx_groups = groups;

4052                group = groups + grpcnt;
4053                ring = group->mrg_rings;
4054                /*
4055                 * The ring can be NULL in the case of aggr. Aggr will
4056                 * have an empty Tx group which will get populated
4057                 * later when pseudo Tx rings are added after
4058                 * mac_register() is done.
4059                 */
4060                if (ring == NULL) {
4061                        ASSERT(mip->mi_state_flags & MIS_IS_AGGR);
4062                        /*
4063                         * pass the group to aggr so it can add Tx
4064                         * rings to the group later.
4065                         */
4066                        cap_rings->mr_gget(mip->mi_driver, rtype, 0, NULL,
4067                            (mac_group_handle_t)group);
4068                        /*
4069                         * Even though there are no rings at this time
4070                         * (rings will come later), set the group
4071                         * state to registered.
4072                         */
4073                        group->mrg_state = MAC_GROUP_STATE_REGISTERED;
4074                } else {
4075                        /*
4076                         * Ring 0 is used as the default one and it could be
4077                         * assigned to a client as well.
4078                         */
4079                        while ((ring->mr_index != 0) && (ring->mr_next != NULL))
4080                                ring = ring->mr_next;
4081                        ASSERT(ring->mr_index == 0);
4082                        mip->mi_default_tx_ring = (mac_ring_handle_t)ring;
4083                }
4084                if (mip->mi_tx_group_type == MAC_GROUP_TYPE_DYNAMIC)
4085                        mip->mi_txrings_avail = group->mrg_cur_count - 1;
4086                        /*
4087                         * The default ring cannot be reserved.
4088                         */
4089                        mip->mi_txrings_rsvd = 1;
4090                /*
4091                 * The default group cannot be reserved. It will be shared
4092                 * by clients that do not have an exclusive group.
```



```
4093                 */
4094                mip->mi_txhwclnt_avail = mip->mi_tx_group_count;
4095                mip->mi_txhwclnt_used = 1;
4096                break;
4097        default:
4098                ASSERT(B_FALSE);
4099        }

4101        if (err != 0)
4102                mac_free_rings(mip, rtype);

4104        return (err);
4105 }

4107 /*
4108  * The ddi interrupt handle could be shared amoung rings. If so, compare
4109  * the new ring's ddi handle with the existing ones and set ddi_shared
4110  * flag.
4111  */
4112 void
4113 mac_compare_ddi_handle(mac_group_t *groups, uint_t grpcnt, mac_ring_t *cring)
4114 {
4115        mac_group_t *group;
4116        mac_ring_t *ring;
4117        ddi_intr_handle_t ddi_handle;
4118        int g;

4120        ddi_handle = cring->mr_info.mri_intr.mi_ddi_handle;
4121        for (g = 0; g < grpcnt; g++) {
4122                group = groups + g;
4123                for (ring = group->mrg_rings; ring != NULL;
4124                    ring = ring->mr_next) {
4125                        if (ring == cring)
4126                                continue;
4127                        if (ring->mr_info.mri_intr.mi_ddi_handle ==
4128                            ddi_handle) {
4129                                if (cring->mr_type == MAC_RING_TYPE_RX &&
4130                                    ring->mr_index == 0 &&
4131                                    !ring->mr_info.mri_intr.mi_ddi_shared) {
4132                                        ring->mr_info.mri_intr.mi_ddi_shared =
4133                                            B_TRUE;
4134                                } else {
4135                                        cring->mr_info.mri_intr.mi_ddi_shared =
4136                                            B_TRUE;
4137                                }
4138                                return;
4139                        }
4140                }
4141        }
4142 }

4144 /*
4145  * Called to free all groups of particular type (RX or TX). It's assumed that
4146  * no clients are using these groups.
4147  */
4148 void
4149 mac_free_rings(mac_impl_t *mip, mac_ring_type_t rtype)
4150 {
4151        mac_group_t *group, *groups;
4152        uint_t group_count;

4154        switch (rtype) {
4155        case MAC_RING_TYPE_RX:
4156                if (mip->mi_rx_groups == NULL)
4157                        return;
```

```
4159                groups = mip->mi_rx_groups;
4160                group_count = mip->mi_rx_group_count;

4162                mip->mi_rx_groups = NULL;
4163                mip->mi_rx_donor_grp = NULL;
4164                mip->mi_rx_group_count = 0;
4165                break;
4166        case MAC_RING_TYPE_TX:
4167                ASSERT(mip->mi_tx_group_count == mip->mi_tx_group_free);

4169                if (mip->mi_tx_groups == NULL)
4170                        return;

4172                groups = mip->mi_tx_groups;
4173                group_count = mip->mi_tx_group_count;

4175                mip->mi_tx_groups = NULL;
4176                mip->mi_tx_group_count = 0;
4177                mip->mi_tx_group_free = 0;
4178                mip->mi_default_tx_ring = NULL;
4179                break;
4180        default:
4181                ASSERT(B_FALSE);
4182        }

4184        for (group = groups; group != NULL; group = group->mrg_next) {
4185                mac_ring_t *ring;

4187                if (group->mrg_cur_count == 0)
4188                        continue;

4190                ASSERT(group->mrg_rings != NULL);

4192                while ((ring = group->mrg_rings) != NULL) {
4193                        group->mrg_rings = ring->mr_next;
4194                        mac_ring_free(mip, ring);
4195                }
4196        }

4198        /* Free all the cached rings */
4199        mac_ring_freeall(mip);
4200        /* Free the block of group data strutures */
4201        kmem_free(groups, sizeof (mac_group_t) * (group_count + 1));
4202 }

4204 /*
4205  * Associate a MAC address with a receive group.
4206  *
4207  * The return value of this function should always be checked properly, because
4208  * any type of failure could cause unexpected results. A group can be added
4209  * or removed with a MAC address only after it has been reserved. Ideally,
4210  * a successful reservation always leads to calling mac_group_addmac() to
4211  * steer desired traffic. Failure of adding an unicast MAC address doesn't
4212  * always imply that the group is functioning abnormally.
4213  *
4214  * Currently this function is called everywhere, and it reflects assumptions
4215  * about MAC addresses in the implementation. CR 6735196.
4216  */
4217 int
4218 mac_group_addmac(mac_group_t *group, const uint8_t *addr)
4219 {
4220        ASSERT(group->mrg_type == MAC_RING_TYPE_RX);
4221        ASSERT(group->mrg_info.mgi_addmac != NULL);

4223        return (group->mrg_info.mgi_addmac(group->mrg_info.mgi_driver, addr));
4224 }
```



```
4226 /*
4227  * Remove the association between MAC address and receive group.
4228  */
4229 int
4230 mac_group_remmac(mac_group_t *group, const uint8_t *addr)
4231 {
4232        ASSERT(group->mrg_type == MAC_RING_TYPE_RX);
4233        ASSERT(group->mrg_info.mgi_remmac != NULL);

4235        return (group->mrg_info.mgi_remmac(group->mrg_info.mgi_driver, addr));
4236 }

4238 /*
4239  * This is the entry point for packets transmitted through the bridging code.
4240  * If no bridge is in place, MAC_RING_TX transmits using tx ring. The 'rh'
4241  * pointer may be NULL to select the default ring.
4242  */
4243 mblk_t *
4244 mac_bridge_tx(mac_impl_t *mip, mac_ring_handle_t rh, mblk_t *mp)
4245 {
4246        mac_handle_t mh;

4248        /*
4249         * Once we take a reference on the bridge link, the bridge
4250         * module itself can't unload, so the callback pointers are
4251         * stable.
4252         */
4253        mutex_enter(&mip->mi_bridge_lock);
4254        if ((mh = mip->mi_bridge_link) != NULL)
4255                mac_bridge_ref_cb(mh, B_TRUE);
4256        mutex_exit(&mip->mi_bridge_lock);
4257        if (mh == NULL) {
4258                MAC_RING_TX(mip, rh, mp, mp);
4259        } else {
4260                mp = mac_bridge_tx_cb(mh, rh, mp);
4261                mac_bridge_ref_cb(mh, B_FALSE);
4262        }

4264        return (mp);
4265 }

4267 /*
4268  * Find a ring from its index.
4269  */
4270 mac_ring_handle_t
4271 mac_find_ring(mac_group_handle_t gh, int index)
4272 {
4273        mac_group_t *group = (mac_group_t *)gh;
4274        mac_ring_t *ring = group->mrg_rings;

4276        for (ring = group->mrg_rings; ring != NULL; ring = ring->mr_next)
4277                if (ring->mr_index == index)
4278                        break;

4280        return ((mac_ring_handle_t)ring);
4281 }
4282 /*
4283  * Add a ring to an existing group.
4284  *
4285  * The ring must be either passed directly (for example if the ring
4286  * movement is initiated by the framework), or specified through a driver
4287  * index (for example when the ring is added by the driver.
4288  *
4289  * The caller needs to call mac_perim_enter() before calling this function.
4290  */
```

```
4291 int
4292 i_mac_group_add_ring(mac_group_t *group, mac_ring_t *ring, int index)
4293 {
4294         mac_impl_t *mip = (mac_impl_t *)group->mrg_mh;
4295         mac_capab_rings_t *cap_rings;
4296         boolean_t driver_call = (ring == NULL);
4297         mac_group_type_t group_type;
4298         int ret = 0;
4299         flow_entry_t *flent;

4301         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

4303         switch (group->mrg_type) {
4304         case MAC_RING_TYPE_RX:
4305                 cap_rings = &mip->mi_rx_rings_cap;
4306                 group_type = mip->mi_rx_group_type;
4307                 break;
4308         case MAC_RING_TYPE_TX:
4309                 cap_rings = &mip->mi_tx_rings_cap;
4310                 group_type = mip->mi_tx_group_type;
4311                 break;
4312         default:
4313                 ASSERT(B_FALSE);
4314         }

4316         /*
4317          * There should be no ring with the same ring index in the target
4318          * group.
4319          */
4320         ASSERT(mac_find_ring((mac_group_handle_t)group,
4321             driver_call ? index : ring->mr_index) == NULL);

4323         if (driver_call) {
4324                 /*
4325                  * The function is called as a result of a request from
4326                  * a driver to add a ring to an existing group, for example
4327                  * from the aggregation driver. Allocate a new mac_ring_t
4328                  * for that ring.
4329                  */
4330                 ring = mac_init_ring(mip, group, index, cap_rings);
4331                 ASSERT(group->mrg_state > MAC_GROUP_STATE_UNINIT);
4332         } else {
4333                 /*
4334                  * The function is called as a result of a MAC layer request
4335                  * to add a ring to an existing group. In this case the
4336                  * ring is being moved between groups, which requires
4337                  * the underlying driver to support dynamic grouping,
4338                  * and the mac_ring_t already exists.
4339                  */
4340                 ASSERT(group_type == MAC_GROUP_TYPE_DYNAMIC);
4341                 ASSERT(group->mrg_driver == NULL ||
4342                     cap_rings->mr_gaddring != NULL);
4343                 ASSERT(ring->mr_gh == NULL);
4344         }

4346         /*
4347          * At this point the ring should not be in use, and it should be
4348          * of the right for the target group.
4349          */
4350         ASSERT(ring->mr_state < MR_INUSE);
4351         ASSERT(ring->mr_srs == NULL);
4352         ASSERT(ring->mr_type == group->mrg_type);

4354         if (!driver_call) {
4355                 /*
4356                  * Add the driver level hardware ring if the process was not
```



```
4357                  * initiated by the driver, and the target group is not the
4358                  * group.
4359                  */
4360                 if (group->mrg_driver != NULL) {
4361                         cap_rings->mr_gaddring(group->mrg_driver,
4362                             ring->mr_driver, ring->mr_type);
4363                 }

4365                 /*
4366                  * Insert the ring ahead existing rings.
4367                  */
4368                 ring->mr_next = group->mrg_rings;
4369                 group->mrg_rings = ring;
4370                 ring->mr_gh = (mac_group_handle_t)group;
4371                 group->mrg_cur_count++;
4372         }

4374         /*
4375          * If the group has not been actively used, we're done.
4376          */
4377         if (group->mrg_index != -1 &&
4378             group->mrg_state < MAC_GROUP_STATE_RESERVED)
4379                 return (0);

4381         /*
4382          * Start the ring if needed. Failure causes to undo the grouping action.
4383          */
4384         if (ring->mr_state != MR_INUSE) {
4385                 if ((ret = mac_start_ring(ring)) != 0) {
4386                         if (!driver_call) {
4387                                 cap_rings->mr_gremring(group->mrg_driver,
4388                                     ring->mr_driver, ring->mr_type);
4389                         }
4390                         group->mrg_cur_count--;
4391                         group->mrg_rings = ring->mr_next;

4393                         ring->mr_gh = NULL;

4395                         if (driver_call)
4396                                 mac_ring_free(mip, ring);

4398                         return (ret);
4399                 }
4400         }

4402         /*
4403          * Set up SRS/SR according to the ring type.
4404          */
4405         switch (ring->mr_type) {
4406         case MAC_RING_TYPE_RX:
4407                 /*
4408                  * Setup SRS on top of the new ring if the group is
4409                  * reserved for someones exclusive use.
4410                  */
4411                 if (group->mrg_state == MAC_GROUP_STATE_RESERVED) {
4412                         mac_client_impl_t *mcip;

4414                         mcip = MAC_GROUP_ONLY_CLIENT(group);
4415                         /*
4416                          * Even though this group is reserved we migth still
4417                          * have multiple clients, i.e a VLAN shares the
4418                          * group with the primary mac client.
4419                          */
4420                         if (mcip != NULL) {
4421                                 flent = mcip->mci_flent;
4422                                 ASSERT(flent->fe_rx_srs_cnt > 0);
```

```
4423                                mac_rx_srs_group_setup(mcip, flent, SRST_LINK);
4424                                mac_fanout_setup(mcip, flent,
4425                                    MCIP_RESOURCE_PROPS(mcip), mac_rx_deliver,
4426                                    mcip, NULL, NULL);
4427                        } else {
4428                                ring->mr_classify_type = MAC_SW_CLASSIFIER;
4429                        }
4430                }
4431                break;
4432        case MAC_RING_TYPE_TX:
4433        {
4434                mac_grp_client_t        *mgcp = group->mrg_clients;
4435                mac_client_impl_t       *mcip;
4436                mac_soft_ring_set_t     *mac_srs;
4437                mac_srs_tx_t            *tx;

4439                if (MAC_GROUP_NO_CLIENT(group)) {
4440                        if (ring->mr_state == MR_INUSE)
4441                                mac_stop_ring(ring);
4442                        ring->mr_flag = 0;
4443                        break;
4444                }
4445                /*
4446                 * If the rings are being moved to a group that has
4447                 * clients using it, then add the new rings to the
4448                 * clients SRS.
4449                 */
4450                while (mgcp != NULL) {
4451                        boolean_t       is_aggr;

4453                        mcip = mgcp->mgc_client;
4454                        flent = mcip->mci_flent;
4455                        is_aggr = (mcip->mci_state_flags & MCIS_IS_AGGR);
4456                        mac_srs = MCIP_TX_SRS(mcip);
4457                        tx = &mac_srs->srs_tx;
4458                        mac_tx_client_quiesce((mac_client_handle_t)mcip);
4459                        /*
4460                         * If we are  growing from 1 to multiple rings.
4461                         */
4462                        if (tx->st_mode == SRS_TX_BW ||
4463                            tx->st_mode == SRS_TX_SERIALIZE ||
4464                            tx->st_mode == SRS_TX_DEFAULT) {
4465                                mac_ring_t      *tx_ring = tx->st_arg2;

4467                                tx->st_arg2 = NULL;
4468                                mac_tx_srs_stat_recreate(mac_srs, B_TRUE);
4469                                mac_tx_srs_add_ring(mac_srs, tx_ring);
4470                                if (mac_srs->srs_type & SRST_BW_CONTROL) {
4471                                        tx->st_mode = is_aggr ? SRS_TX_BW_AGGR :
4472                                            SRS_TX_BW_FANOUT;
4473                                } else {
4474                                        tx->st_mode = is_aggr ? SRS_TX_AGGR :
4475                                            SRS_TX_FANOUT;
4476                                }
4477                                tx->st_func = mac_tx_get_func(tx->st_mode);
4478                        }
4479                        mac_tx_srs_add_ring(mac_srs, ring);
4480                        mac_fanout_setup(mcip, flent, MCIP_RESOURCE_PROPS(mcip),
4481                            mac_rx_deliver, mcip, NULL, NULL);
4482                        mac_tx_client_restart((mac_client_handle_t)mcip);
4483                        mgcp = mgcp->mgc_next;
4484                }
4485                break;
4486        }
4487        default:
4488                ASSERT(B_FALSE);
```



```
4489        }
4490        /*
4491         * For aggr, the default ring will be NULL to begin with. If it
4492         * is NULL, then pick the first ring that gets added as the
4493         * default ring. Any ring in an aggregation can be removed at
4494         * any time (by the user action of removing a link) and if the
4495         * current default ring gets removed, then a new one gets
4496         * picked (see i_mac_group_rem_ring()).
4497         */
4498        if (mip->mi_state_flags & MIS_IS_AGGR &&
4499            mip->mi_default_tx_ring == NULL &&
4500            ring->mr_type == MAC_RING_TYPE_TX) {
4501                mip->mi_default_tx_ring = (mac_ring_handle_t)ring;
4502        }

4504        MAC_RING_UNMARK(ring, MR_INCIPIENT);
4505        return (0);
4506 }

4508 /*
4509  * Remove a ring from it's current group. MAC internal function for dynamic
4510  * grouping.
4511  *
4512  * The caller needs to call mac_perim_enter() before calling this function.
4513  */
4514 void
4515 i_mac_group_rem_ring(mac_group_t *group, mac_ring_t *ring,
4516     boolean_t driver_call)
4517 {
4518        mac_impl_t *mip = (mac_impl_t *)group->mrg_mh;
4519        mac_capab_rings_t *cap_rings = NULL;
4520        mac_group_type_t group_type;

4522        ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

4524        ASSERT(mac_find_ring((mac_group_handle_t)group,
4525            ring->mr_index) == (mac_ring_handle_t)ring);
4526        ASSERT((mac_group_t *)ring->mr_gh == group);
4527        ASSERT(ring->mr_type == group->mrg_type);

4529        if (ring->mr_state == MR_INUSE)
4530                mac_stop_ring(ring);
4531        switch (ring->mr_type) {
4532        case MAC_RING_TYPE_RX:
4533                group_type = mip->mi_rx_group_type;
4534                cap_rings = &mip->mi_rx_rings_cap;

4536                /*
4537                 * Only hardware classified packets hold a reference to the
4538                 * ring all the way up the Rx path. mac_rx_srs_remove()
4539                 * will take care of quiescing the Rx path and removing the
4540                 * SRS. The software classified path neither holds a reference
4541                 * nor any association with the ring in mac_rx.
4542                 */
4543                if (ring->mr_srs != NULL) {
4544                        mac_rx_srs_remove(ring->mr_srs);
4545                        ring->mr_srs = NULL;
4546                }

4548                break;
4549        case MAC_RING_TYPE_TX:
4550        {
4551                mac_grp_client_t        *mgcp;
4552                mac_client_impl_t       *mcip;
4553                mac_soft_ring_set_t     *mac_srs;
4554                mac_srs_tx_t            *tx;
```

```
4555                mac_ring_t              *rem_ring;
4556                mac_group_t             *defgrp;
4557                uint_t                  ring_info = 0;

4559                /*
4560                 * For TX this function is invoked in three
4561                 * cases:
4562                 *
4563                 * 1) In the case of a failure during the
4564                 * initial creation of a group when a share is
4565                 * associated with a MAC client. So the SRS is not
4566                 * yet setup, and will be setup later after the
4567                 * group has been reserved and populated.
4568                 *
4569                 * 2) From mac_release_tx_group() when freeing
4570                 * a TX SRS.
4571                 *
4572                 * 3) In the case of aggr, when a port gets removed,
4573                 * the pseudo Tx rings that it exposed gets removed.
4574                 *
4575                 * In the first two cases the SRS and its soft
4576                 * rings are already quiesced.
4577                 */
4578                if (driver_call) {
4579                        mac_client_impl_t *mcip;
4580                        mac_soft_ring_set_t *mac_srs;
4581                        mac_soft_ring_t *sringp;
4582                        mac_srs_tx_t *srs_tx;

4584                        if (mip->mi_state_flags & MIS_IS_AGGR &&
4585                            mip->mi_default_tx_ring ==
4586                            (mac_ring_handle_t)ring) {
4587                                /* pick a new default Tx ring */
4588                                mip->mi_default_tx_ring =
4589                                    (group->mrg_rings != ring) ?
4590                                    (mac_ring_handle_t)group->mrg_rings :
4591                                    (mac_ring_handle_t)(ring->mr_next);
4592                        }
4593                        /* Presently only aggr case comes here */
4594                        if (group->mrg_state != MAC_GROUP_STATE_RESERVED)
4595                                break;

4597                        mcip = MAC_GROUP_ONLY_CLIENT(group);
4598                        ASSERT(mcip != NULL);
4599                        ASSERT(mcip->mci_state_flags & MCIS_IS_AGGR);
4600                        mac_srs = MCIP_TX_SRS(mcip);
4601                        ASSERT(mac_srs->srs_tx.st_mode == SRS_TX_AGGR ||
4602                            mac_srs->srs_tx.st_mode == SRS_TX_BW_AGGR);
4603                        srs_tx = &mac_srs->srs_tx;
4604                        /*
4605                         * Wakeup any callers blocked on this
4606                         * Tx ring due to flow control.
4607                         */
4608                        sringp = srs_tx->st_soft_rings[ring->mr_index];
4609                        ASSERT(sringp != NULL);
4610                        mac_tx_invoke_callbacks(mcip, (mac_tx_cookie_t)sringp);
4611                        mac_tx_client_quiesce((mac_client_handle_t)mcip);
4612                        mac_tx_srs_del_ring(mac_srs, ring);
4613                        mac_tx_client_restart((mac_client_handle_t)mcip);
4614                        break;
4615                }
4616                ASSERT(ring != (mac_ring_t *)mip->mi_default_tx_ring);
4617                group_type = mip->mi_tx_group_type;
4618                cap_rings = &mip->mi_tx_rings_cap;
4619                /*
4620                 * See if we need to take it out of the MAC clients using
```



```
4621                 * this group
4622                 */
4623                if (MAC_GROUP_NO_CLIENT(group))
4624                        break;
4625                mgcp = group->mrg_clients;
4626                defgrp = MAC_DEFAULT_TX_GROUP(mip);
4627                while (mgcp != NULL) {
4628                        mcip = mgcp->mgc_client;
4629                        mac_srs = MCIP_TX_SRS(mcip);
4630                        tx = &mac_srs->srs_tx;
4631                        mac_tx_client_quiesce((mac_client_handle_t)mcip);
4632                        /*
4633                         * If we are here when removing rings from the
4634                         * defgroup, mac_reserve_tx_ring would have
4635                         * already deleted the ring from the MAC
4636                         * clients in the group.
4637                         */
4638                        if (group != defgrp) {
4639                                mac_tx_invoke_callbacks(mcip,
4640                                    (mac_tx_cookie_t)
4641                                    mac_tx_srs_get_soft_ring(mac_srs, ring));
4642                                mac_tx_srs_del_ring(mac_srs, ring);
4643                        }
4644                        /*
4645                         * Additionally, if  we are left with only
4646                         * one ring in the group after this, we need
4647                         * to modify the mode etc. to. (We haven't
4648                         * yet taken the ring out, so we check with 2).
4649                         */
4650                        if (group->mrg_cur_count == 2) {
4651                                if (ring->mr_next == NULL)
4652                                        rem_ring = group->mrg_rings;
4653                                else
4654                                        rem_ring = ring->mr_next;
4655                                mac_tx_invoke_callbacks(mcip,
4656                                    (mac_tx_cookie_t)
4657                                    mac_tx_srs_get_soft_ring(mac_srs,
4658                                    rem_ring));
4659                                mac_tx_srs_del_ring(mac_srs, rem_ring);
4660                                if (rem_ring->mr_state != MR_INUSE) {
4661                                        (void) mac_start_ring(rem_ring);
4662                                }
4663                                tx->st_arg2 = (void *)rem_ring;
4664                                mac_tx_srs_stat_recreate(mac_srs, B_FALSE);
4665                                ring_info = mac_hwring_getinfo(
4666                                    (mac_ring_handle_t)rem_ring);
4667                                /*
4668                                 * We are  shrinking from multiple
4669                                 * to 1 ring.
4670                                 */
4671                                if (mac_srs->srs_type & SRST_BW_CONTROL) {
4672                                        tx->st_mode = SRS_TX_BW;
4673                                } else if (mac_tx_serialize ||
4674                                    (ring_info & MAC_RING_TX_SERIALIZE)) {
4675                                        tx->st_mode = SRS_TX_SERIALIZE;
4676                                } else {
4677                                        tx->st_mode = SRS_TX_DEFAULT;
4678                                }
4679                                tx->st_func = mac_tx_get_func(tx->st_mode);
4680                        }
4681                        mac_tx_client_restart((mac_client_handle_t)mcip);
4682                        mgcp = mgcp->mgc_next;
4683                }
4684                break;
4685        }
4686        default:
```

```
4687                    ASSERT(B_FALSE);
4688            }

4690            /*
4691             * Remove the ring from the group.
4692             */
4693            if (ring == group->mrg_rings)
4694                    group->mrg_rings = ring->mr_next;
4695            else {
4696                    mac_ring_t *pre;

4698                    pre = group->mrg_rings;
4699                    while (pre->mr_next != ring)
4700                            pre = pre->mr_next;
4701                    pre->mr_next = ring->mr_next;
4702            }
4703            group->mrg_cur_count--;

4705            if (!driver_call) {
4706                    ASSERT(group_type == MAC_GROUP_TYPE_DYNAMIC);
4707                    ASSERT(group->mrg_driver == NULL ||
4708                        cap_rings->mr_gremring != NULL);

4710                    /*
4711                     * Remove the driver level hardware ring.
4712                     */
4713                    if (group->mrg_driver != NULL) {
4714                            cap_rings->mr_gremring(group->mrg_driver,
4715                                ring->mr_driver, ring->mr_type);
4716                    }
4717            }

4719            ring->mr_gh = NULL;
4720            if (driver_call)
4721                    mac_ring_free(mip, ring);
4722            else
4723                    ring->mr_flag = 0;
4724    }

4726    /*
4727     * Move a ring to the target group. If needed, remove the ring from the group
4728     * that it currently belongs to.
4729     *
4730     * The caller need to enter MAC's perimeter by calling mac_perim_enter().
4731     */
4732    static int
4733    mac_group_mov_ring(mac_impl_t *mip, mac_group_t *d_group, mac_ring_t *ring)
4734    {
4735            mac_group_t *s_group = (mac_group_t *)ring->mr_gh;
4736            int rv;

4738            ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));
4739            ASSERT(d_group != NULL);
4740            ASSERT(s_group->mrg_mh == d_group->mrg_mh);

4742            if (s_group == d_group)
4743                    return (0);

4745            /*
4746             * Remove it from current group first.
4747             */
4748            if (s_group != NULL)
4749                    i_mac_group_rem_ring(s_group, ring, B_FALSE);

4751            /*
4752             * Add it to the new group.
```



```
4753             */
4754            rv = i_mac_group_add_ring(d_group, ring, 0);
4755            if (rv != 0) {
4756                    /*
4757                     * Failed to add ring back to source group. If
4758                     * that fails, the ring is stuck in limbo, log message.
4759                     */
4760                    if (i_mac_group_add_ring(s_group, ring, 0)) {
4761                            cmn_err(CE_WARN, "%s: failed to move ring %p\n",
4762                                mip->mi_name, (void *)ring);
4763                    }
4764            }

4766            return (rv);
4767    }

4769    /*
4770     * Find a MAC address according to its value.
4771     */
4772    mac_address_t *
4773    mac_find_macaddr(mac_impl_t *mip, uint8_t *mac_addr)
4774    {
4775            mac_address_t *map;

4777            ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

4779            for (map = mip->mi_addresses; map != NULL; map = map->ma_next) {
4780                    if (bcmp(mac_addr, map->ma_addr, map->ma_len) == 0)
4781                            break;
4782            }

4784            return (map);
4785    }

4787    /*
4788     * Check whether the MAC address is shared by multiple clients.
4789     */
4790    boolean_t
4791    mac_check_macaddr_shared(mac_address_t *map)
4792    {
4793            ASSERT(MAC_PERIM_HELD((mac_handle_t)map->ma_mip));

4795            return (map->ma_nusers > 1);
4796    }

4798    /*
4799     * Remove the specified MAC address from the MAC address list and free it.
4800     */
4801    static void
4802    mac_free_macaddr(mac_address_t *map)
4803    {
4804            mac_impl_t *mip = map->ma_mip;

4806            ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));
4807            ASSERT(mip->mi_addresses != NULL);

4809            map = mac_find_macaddr(mip, map->ma_addr);

4811            ASSERT(map != NULL);
4812            ASSERT(map->ma_nusers == 0);

4814            if (map == mip->mi_addresses) {
4815                    mip->mi_addresses = map->ma_next;
4816            } else {
4817                    mac_address_t *pre;
```

```
4819                 pre = mip->mi_addresses;
4820                 while (pre->ma_next != map)
4821                         pre = pre->ma_next;
4822                 pre->ma_next = map->ma_next;
4823         }

4825         kmem_free(map, sizeof (mac_address_t));
4826 }

4828 /*
4829  * Add a MAC address reference for a client. If the desired MAC address
4830  * exists, add a reference to it. Otherwise, add the new address by adding
4831  * it to a reserved group or setting promiscuous mode. Won't try different
4832  * group is the group is non-NULL, so the caller must explictly share
4833  * default group when needed.
4834  *
4835  * Note, the primary MAC address is initialized at registration time, so
4836  * to add it to default group only need to activate it if its reference
4837  * count is still zero. Also, some drivers may not have advertised RINGS
4838  * capability.
4839  */
4840 int
4841 mac_add_macaddr(mac_impl_t *mip, mac_group_t *group, uint8_t *mac_addr,
4842     boolean_t use_hw)
4843 {
4844         mac_address_t *map;
4845         int err = 0;
4846         boolean_t allocated_map = B_FALSE;

4848         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

4850         map = mac_find_macaddr(mip, mac_addr);

4852         /*
4853          * If the new MAC address has not been added. Allocate a new one
4854          * and set it up.
4855          */
4856         if (map == NULL) {
4857                 map = kmem_zalloc(sizeof (mac_address_t), KM_SLEEP);
4858                 map->ma_len = mip->mi_type->mt_addr_length;
4859                 bcopy(mac_addr, map->ma_addr, map->ma_len);
4860                 map->ma_nusers = 0;
4861                 map->ma_group = group;
4862                 map->ma_mip = mip;

4864                 /* add the new MAC address to the head of the address list */
4865                 map->ma_next = mip->mi_addresses;
4866                 mip->mi_addresses = map;

4868                 allocated_map = B_TRUE;
4869         }

4871         ASSERT(map->ma_group == NULL || map->ma_group == group);
4872         if (map->ma_group == NULL)
4873                 map->ma_group = group;

4875         /*
4876          * If the MAC address is already in use, simply account for the
4877          * new client.
4878          */
4879         if (map->ma_nusers++ > 0)
4880                 return (0);

4882         /*
4883          * Activate this MAC address by adding it to the reserved group.
4884          */
```



```
4885         if (group != NULL) {
4886                 err = mac_group_addmac(group, (const uint8_t *)mac_addr);
4887                 if (err == 0) {
4888                         map->ma_type = MAC_ADDRESS_TYPE_UNICAST_CLASSIFIED;
4889                         return (0);
4890                 }
4891         }

4893         /*
4894          * The MAC address addition failed. If the client requires a
4895          * hardware classified MAC address, fail the operation.
4896          */
4897         if (use_hw) {
4898                 err = ENOSPC;
4899                 goto bail;
4900         }

4902         /*
4903          * Try promiscuous mode.
4904          *
4905          * For drivers that don't advertise RINGS capability, do
4906          * nothing for the primary address.
4907          */
4908         if ((group == NULL) &&
4909             (bcmp(map->ma_addr, mip->mi_addr, map->ma_len) == 0)) {
4910                 map->ma_type = MAC_ADDRESS_TYPE_UNICAST_CLASSIFIED;
4911                 return (0);
4912         }

4914         /*
4915          * Enable promiscuous mode in order to receive traffic
4916          * to the new MAC address.
4917          */
4918         if ((err = i_mac_promisc_set(mip, B_TRUE)) == 0) {
4919                 map->ma_type = MAC_ADDRESS_TYPE_UNICAST_PROMISC;
4920                 return (0);
4921         }

4923         /*
4924          * Free the MAC address that could not be added. Don't free
4925          * a pre-existing address, it could have been the entry
4926          * for the primary MAC address which was pre-allocated by
4927          * mac_init_macaddr(), and which must remain on the list.
4928          */
4929 bail:
4930         map->ma_nusers--;
4931         if (allocated_map)
4932                 mac_free_macaddr(map);
4933         return (err);
4934 }

4936 /*
4937  * Remove a reference to a MAC address. This may cause to remove the MAC
4938  * address from an associated group or to turn off promiscuous mode.
4939  * The caller needs to handle the failure properly.
4940  */
4941 int
4942 mac_remove_macaddr(mac_address_t *map)
4943 {
4944         mac_impl_t *mip = map->ma_mip;
4945         int err = 0;

4947         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

4949         ASSERT(map == mac_find_macaddr(mip, map->ma_addr));
```

```
4951         /*
4952          * If it's not the last client using this MAC address, only update
4953          * the MAC clients count.
4954          */
4955         if (--map->ma_nusers > 0)
4956                 return (0);

4958         /*
4959          * The MAC address is no longer used by any MAC client, so remove
4960          * it from its associated group, or turn off promiscuous mode
4961          * if it was enabled for the MAC address.
4962          */
4963         switch (map->ma_type) {
4964         case MAC_ADDRESS_TYPE_UNICAST_CLASSIFIED:
4965                 /*
4966                  * Don't free the preset primary address for drivers that
4967                  * don't advertise RINGS capability.
4968                  */
4969                 if (map->ma_group == NULL)
4970                         return (0);

4972                 err = mac_group_remmac(map->ma_group, map->ma_addr);
4973                 if (err == 0)
4974                         map->ma_group = NULL;
4975                 break;
4976         case MAC_ADDRESS_TYPE_UNICAST_PROMISC:
4977                 err = i_mac_promisc_set(mip, B_FALSE);
4978                 break;
4979         default:
4980                 ASSERT(B_FALSE);
4981         }

4983         if (err != 0)
4984                 return (err);

4986         /*
4987          * We created MAC address for the primary one at registration, so we
4988          * won't free it here. mac_fini_macaddr() will take care of it.
4989          */
4990         if (bcmp(map->ma_addr, mip->mi_addr, map->ma_len) != 0)
4991                 mac_free_macaddr(map);

4993         return (0);
4994 }

4996 /*
4997  * Update an existing MAC address. The caller need to make sure that the new
4998  * value has not been used.
4999  */
5000 int
5001 mac_update_macaddr(mac_address_t *map, uint8_t *mac_addr)
5002 {
5003         mac_impl_t *mip = map->ma_mip;
5004         int err = 0;

5006         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));
5007         ASSERT(mac_find_macaddr(mip, mac_addr) == NULL);

5009         switch (map->ma_type) {
5010         case MAC_ADDRESS_TYPE_UNICAST_CLASSIFIED:
5011                 /*
5012                  * Update the primary address for drivers that are not
5013                  * RINGS capable.
5014                  */
5015                 if (mip->mi_rx_groups == NULL) {
5016                         err = mip->mi_unicst(mip->mi_driver, (const uint8_t *)
```



```
5017                             mac_addr);
5018                         if (err != 0)
5019                                 return (err);
5020                         break;
5021                 }

5023                 /*
5024                  * If this MAC address is not currently in use,
5025                  * simply break out and update the value.
5026                  */
5027                 if (map->ma_nusers == 0)
5028                         break;

5030                 /*
5031                  * Need to replace the MAC address associated with a group.
5032                  */
5033                 err = mac_group_remmac(map->ma_group, map->ma_addr);
5034                 if (err != 0)
5035                         return (err);

5037                 err = mac_group_addmac(map->ma_group, mac_addr);

5039                 /*
5040                  * Failure hints hardware error. The MAC layer needs to
5041                  * have error notification facility to handle this.
5042                  * Now, simply try to restore the value.
5043                  */
5044                 if (err != 0)
5045                         (void) mac_group_addmac(map->ma_group, map->ma_addr);

5047                 break;
5048         case MAC_ADDRESS_TYPE_UNICAST_PROMISC:
5049                 /*
5050                  * Need to do nothing more if in promiscuous mode.
5051                  */
5052                 break;
5053         default:
5054                 ASSERT(B_FALSE);
5055         }

5057         /*
5058          * Successfully replaced the MAC address.
5059          */
5060         if (err == 0)
5061                 bcopy(mac_addr, map->ma_addr, map->ma_len);

5063         return (err);
5064 }

5066 /*
5067  * Freshen the MAC address with new value. Its caller must have updated the
5068  * hardware MAC address before calling this function.
5069  * This funcitons is supposed to be used to handle the MAC address change
5070  * notification from underlying drivers.
5071  */
5072 void
5073 mac_freshen_macaddr(mac_address_t *map, uint8_t *mac_addr)
5074 {
5075         mac_impl_t *mip = map->ma_mip;

5077         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));
5078         ASSERT(mac_find_macaddr(mip, mac_addr) == NULL);

5080         /*
5081          * Freshen the MAC address with new value.
5082          */
```

```
5083         bcopy(mac_addr, map->ma_addr, map->ma_len);
5084         bcopy(mac_addr, mip->mi_addr, map->ma_len);

5086         /*
5087          * Update all MAC clients that share this MAC address.
5088          */
5089         mac_unicast_update_clients(mip, map);
5090 }

5092 /*
5093  * Set up the primary MAC address.
5094  */
5095 void
5096 mac_init_macaddr(mac_impl_t *mip)
5097 {
5098         mac_address_t *map;

5100         /*
5101          * The reference count is initialized to zero, until it's really
5102          * activated.
5103          */
5104         map = kmem_zalloc(sizeof (mac_address_t), KM_SLEEP);
5105         map->ma_len = mip->mi_type->mt_addr_length;
5106         bcopy(mip->mi_addr, map->ma_addr, map->ma_len);

5108         /*
5109          * If driver advertises RINGS capability, it shouldn't have initialized
5110          * its primary MAC address. For other drivers, including VNIC, the
5111          * primary address must work after registration.
5112          */
5113         if (mip->mi_rx_groups == NULL)
5114                 map->ma_type = MAC_ADDRESS_TYPE_UNICAST_CLASSIFIED;

5116         map->ma_mip = mip;

5118         mip->mi_addresses = map;
5119 }

5121 /*
5122  * Clean up the primary MAC address. Note, only one primary MAC address
5123  * is allowed. All other MAC addresses must have been freed appropriately.
5124  */
5125 void
5126 mac_fini_macaddr(mac_impl_t *mip)
5127 {
5128         mac_address_t *map = mip->mi_addresses;

5130         if (map == NULL)
5131                 return;

5133         /*
5134          * If mi_addresses is initialized, there should be exactly one
5135          * entry left on the list with no users.
5136          */
5137         ASSERT(map->ma_nusers == 0);
5138         ASSERT(map->ma_next == NULL);

5140         kmem_free(map, sizeof (mac_address_t));
5141         mip->mi_addresses = NULL;
5142 }

5144 /*
5145  * Logging related functions.
5146  *
5147  * Note that Kernel statistics have been extended to maintain fine
5148  * granularity of statistics viz. hardware lane, software lane, fanout
```



```
5149  * stats etc. However, extended accounting continues to support only
5150  * aggregate statistics like before.
5151  */

5153 /* Write the flow description to a netinfo_t record */
5154 static netinfo_t *
5155 mac_write_flow_desc(flow_entry_t *flent, mac_client_impl_t *mcip)
5156 {
5157         netinfo_t               *ninfo;
5158         net_desc_t              *ndesc;
5159         flow_desc_t             *fdesc;
5160         mac_resource_props_t    *mrp;

5162         ninfo = kmem_zalloc(sizeof (netinfo_t), KM_NOSLEEP);
5163         if (ninfo == NULL)
5164                 return (NULL);
5165         ndesc = kmem_zalloc(sizeof (net_desc_t), KM_NOSLEEP);
5166         if (ndesc == NULL) {
5167                 kmem_free(ninfo, sizeof (netinfo_t));
5168                 return (NULL);
5169         }

5171         /*
5172          * Grab the fe_lock to see a self-consistent fe_flow_desc.
5173          * Updates to the fe_flow_desc are done under the fe_lock
5174          */
5175         mutex_enter(&flent->fe_lock);
5176         fdesc = &flent->fe_flow_desc;
5177         mrp = &flent->fe_resource_props;

5179         ndesc->nd_name = flent->fe_flow_name;
5180         ndesc->nd_devname = mcip->mci_name;
5181         bcopy(fdesc->fd_src_mac, ndesc->nd_ehost, ETHERADDRL);
5182         bcopy(fdesc->fd_dst_mac, ndesc->nd_edest, ETHERADDRL);
5183         ndesc->nd_sap = htonl(fdesc->fd_sap);
5184         ndesc->nd_isv4 = (uint8_t)fdesc->fd_ipversion == IPV4_VERSION;
5185         ndesc->nd_bw_limit = mrp->mrp_maxbw;
5186         if (ndesc->nd_isv4) {
5187                 ndesc->nd_saddr[3] = htonl(fdesc->fd_local_addr.s6_addr32[3]);
5188                 ndesc->nd_daddr[3] = htonl(fdesc->fd_remote_addr.s6_addr32[3]);
5189         } else {
5190                 bcopy(&fdesc->fd_local_addr, ndesc->nd_saddr, IPV6_ADDR_LEN);
5191                 bcopy(&fdesc->fd_remote_addr, ndesc->nd_daddr, IPV6_ADDR_LEN);
5192         }
5193         ndesc->nd_sport = htons(fdesc->fd_local_port);
5194         ndesc->nd_dport = htons(fdesc->fd_remote_port);
5195         ndesc->nd_protocol = (uint8_t)fdesc->fd_protocol;
5196         mutex_exit(&flent->fe_lock);

5198         ninfo->ni_record = ndesc;
5199         ninfo->ni_size = sizeof (net_desc_t);
5200         ninfo->ni_type = EX_NET_FLDESC_REC;

5202         return (ninfo);
5203 }

5205 /* Write the flow statistics to a netinfo_t record */
5206 static netinfo_t *
5207 mac_write_flow_stats(flow_entry_t *flent)
5208 {
5209         netinfo_t               *ninfo;
5210         net_stat_t              *nstat;
5211         mac_soft_ring_set_t     *mac_srs;
5212         mac_rx_stats_t          *mac_rx_stat;
5213         mac_tx_stats_t          *mac_tx_stat;
5214         int                     i;
```

```
5216         ninfo = kmem_zalloc(sizeof (netinfo_t), KM_NOSLEEP);
5217         if (ninfo == NULL)
5218                 return (NULL);
5219         nstat = kmem_zalloc(sizeof (net_stat_t), KM_NOSLEEP);
5220         if (nstat == NULL) {
5221                 kmem_free(ninfo, sizeof (netinfo_t));
5222                 return (NULL);
5223         }

5225         nstat->ns_name = flent->fe_flow_name;
5226         for (i = 0; i < flent->fe_rx_srs_cnt; i++) {
5227                 mac_srs = (mac_soft_ring_set_t *)flent->fe_rx_srs[i];
5228                 mac_rx_stat = &mac_srs->srs_rx.sr_stat;

5230                 nstat->ns_ibytes += mac_rx_stat->mrs_intrbytes +
5231                     mac_rx_stat->mrs_pollbytes + mac_rx_stat->mrs_lclbytes;
5232                 nstat->ns_ipackets += mac_rx_stat->mrs_intrcnt +
5233                     mac_rx_stat->mrs_pollcnt + mac_rx_stat->mrs_lclcnt;
5234                 nstat->ns_oerrors += mac_rx_stat->mrs_ierrors;
5235         }

5237         mac_srs = (mac_soft_ring_set_t *)(flent->fe_tx_srs);
5238         if (mac_srs != NULL) {
5239                 mac_tx_stat = &mac_srs->srs_tx.st_stat;

5241                 nstat->ns_obytes = mac_tx_stat->mts_obytes;
5242                 nstat->ns_opackets = mac_tx_stat->mts_opackets;
5243                 nstat->ns_oerrors = mac_tx_stat->mts_oerrors;
5244         }

5246         ninfo->ni_record = nstat;
5247         ninfo->ni_size = sizeof (net_stat_t);
5248         ninfo->ni_type = EX_NET_FLSTAT_REC;

5250         return (ninfo);
5251 }

5253 /* Write the link description to a netinfo_t record */
5254 static netinfo_t *
5255 mac_write_link_desc(mac_client_impl_t *mcip)
5256 {
5257         netinfo_t               *ninfo;
5258         net_desc_t              *ndesc;
5259         flow_entry_t            *flent = mcip->mci_flent;

5261         ninfo = kmem_zalloc(sizeof (netinfo_t), KM_NOSLEEP);
5262         if (ninfo == NULL)
5263                 return (NULL);
5264         ndesc = kmem_zalloc(sizeof (net_desc_t), KM_NOSLEEP);
5265         if (ndesc == NULL) {
5266                 kmem_free(ninfo, sizeof (netinfo_t));
5267                 return (NULL);
5268         }

5270         ndesc->nd_name = mcip->mci_name;
5271         ndesc->nd_devname = mcip->mci_name;
5272         ndesc->nd_isv4 = B_TRUE;
5273         /*
5274          * Grab the fe_lock to see a self-consistent fe_flow_desc.
5275          * Updates to the fe_flow_desc are done under the fe_lock
5276          * after removing the flent from the flow table.
5277          */
5278         mutex_enter(&flent->fe_lock);
5279         bcopy(flent->fe_flow_desc.fd_src_mac, ndesc->nd_ehost, ETHERADDRL);
5280         mutex_exit(&flent->fe_lock);
```



```
5282         ninfo->ni_record = ndesc;
5283         ninfo->ni_size = sizeof (net_desc_t);
5284         ninfo->ni_type = EX_NET_LNDESC_REC;

5286         return (ninfo);
5287 }

5289 /* Write the link statistics to a netinfo_t record */
5290 static netinfo_t *
5291 mac_write_link_stats(mac_client_impl_t *mcip)
5292 {
5293         netinfo_t               *ninfo;
5294         net_stat_t              *nstat;
5295         flow_entry_t            *flent;
5296         mac_soft_ring_set_t     *mac_srs;
5297         mac_rx_stats_t          *mac_rx_stat;
5298         mac_tx_stats_t          *mac_tx_stat;
5299         int                     i;

5301         ninfo = kmem_zalloc(sizeof (netinfo_t), KM_NOSLEEP);
5302         if (ninfo == NULL)
5303                 return (NULL);
5304         nstat = kmem_zalloc(sizeof (net_stat_t), KM_NOSLEEP);
5305         if (nstat == NULL) {
5306                 kmem_free(ninfo, sizeof (netinfo_t));
5307                 return (NULL);
5308         }

5310         nstat->ns_name = mcip->mci_name;
5311         flent = mcip->mci_flent;
5312         if (flent != NULL)  {
5313                 for (i = 0; i < flent->fe_rx_srs_cnt; i++) {
5314                         mac_srs = (mac_soft_ring_set_t *)flent->fe_rx_srs[i];
5315                         mac_rx_stat = &mac_srs->srs_rx.sr_stat;

5317                         nstat->ns_ibytes += mac_rx_stat->mrs_intrbytes +
5318                             mac_rx_stat->mrs_pollbytes +
5319                             mac_rx_stat->mrs_lclbytes;
5320                         nstat->ns_ipackets += mac_rx_stat->mrs_intrcnt +
5321                             mac_rx_stat->mrs_pollcnt + mac_rx_stat->mrs_lclcnt;
5322                         nstat->ns_oerrors += mac_rx_stat->mrs_ierrors;
5323                 }
5324         }

5326         mac_srs = (mac_soft_ring_set_t *)(mcip->mci_flent->fe_tx_srs);
5327         if (mac_srs != NULL) {
5328                 mac_tx_stat = &mac_srs->srs_tx.st_stat;

5330                 nstat->ns_obytes = mac_tx_stat->mts_obytes;
5331                 nstat->ns_opackets = mac_tx_stat->mts_opackets;
5332                 nstat->ns_oerrors = mac_tx_stat->mts_oerrors;
5333         }

5335         ninfo->ni_record = nstat;
5336         ninfo->ni_size = sizeof (net_stat_t);
5337         ninfo->ni_type = EX_NET_LNSTAT_REC;

5339         return (ninfo);
5340 }

5342 typedef struct i_mac_log_state_s {
5343         boolean_t       mi_last;
5344         int             mi_fenable;
5345         int             mi_lenable;
5346         list_t          *mi_list;
```

```
} i_mac_log_state_t;

/*
 * For a given flow, if the description has not been logged before, do it now.
 * If it is a VNIC, then we have collected information about it from the MAC
 * table, so skip it.
 *
 * Called through mac_flow_walk_nolock()
 *
 * Return 0 if successful.
 */
static int
mac_log_flowinfo(flow_entry_t *flent, void *arg)
{
        mac_client_impl_t       *mcip = flent->fe_mcip;
        i_mac_log_state_t       *lstate = arg;
        netinfo_t               *ninfo;

        if (mcip == NULL)
                return (0);

        /*
         * If the name starts with "vnic", and fe_user_generated is true (to
         * exclude the mcast and active flow entries created implicitly for
         * a vnic, it is a VNIC flow.  i.e. vnic1 is a vnic flow,
         * vnic/bge1/mcast1 is not and neither is vnic/bge1/active.
         */
        if (strncasecmp(flent->fe_flow_name, "vnic", 4) == 0 &&
            (flent->fe_type & FLOW_USER) != 0) {
                return (0);
        }

        if (!flent->fe_desc_logged) {
                /*
                 * We don't return error because we want to continue the
                 * walk in case this is the last walk which means we
                 * need to reset fe_desc_logged in all the flows.
                 */
                if ((ninfo = mac_write_flow_desc(flent, mcip)) == NULL)
                        return (0);
                list_insert_tail(lstate->mi_list, ninfo);
                flent->fe_desc_logged = B_TRUE;
        }

        /*
         * Regardless of the error, we want to proceed in case we have to
         * reset fe_desc_logged.
         */
        ninfo = mac_write_flow_stats(flent);
        if (ninfo == NULL)
                return (-1);

        list_insert_tail(lstate->mi_list, ninfo);

        if (mcip != NULL && !(mcip->mci_state_flags & MCIS_DESC_LOGGED))
                flent->fe_desc_logged = B_FALSE;

        return (0);
}

/*
 * Log the description for each mac client of this mac_impl_t, if it
 * hasn't already been done. Additionally, log statistics for the link as
 * well. Walk the flow table and log information for each flow as well.
 * If it is the last walk (mci_last), then we turn off mci_desc_logged (and
 * also fe_desc_logged, if flow logging is on) since we want to log the
```



```
 * description if and when logging is restarted.
 *
 * Return 0 upon success or -1 upon failure
 */
static int
i_mac_impl_log(mac_impl_t *mip, i_mac_log_state_t *lstate)
{
        mac_client_impl_t       *mcip;
        netinfo_t               *ninfo;

        i_mac_perim_enter(mip);
        /*
         * Only walk the client list for NIC and etherstub
         */
        if ((mip->mi_state_flags & MIS_DISABLED) ||
            ((mip->mi_state_flags & MIS_IS_VNIC) &&
            (mac_get_lower_mac_handle((mac_handle_t)mip) != NULL))) {
                i_mac_perim_exit(mip);
                return (0);
        }

        for (mcip = mip->mi_clients_list; mcip != NULL;
            mcip = mcip->mci_client_next) {
                if (!MCIP_DATAPATH_SETUP(mcip))
                        continue;
                if (lstate->mi_lenable) {
                        if (!(mcip->mci_state_flags & MCIS_DESC_LOGGED)) {
                                ninfo = mac_write_link_desc(mcip);
                                if (ninfo == NULL) {
                                /*
                                 * We can't terminate it if this is the last
                                 * walk, else there might be some links with
                                 * mi_desc_logged set to true, which means
                                 * their description won't be logged the next
                                 * time logging is started (similarly for the
                                 * flows within such links). We can continue
                                 * without walking the flow table (i.e. to
                                 * set fe_desc_logged to false) because we
                                 * won't have written any flow stuff for this
                                 * link as we haven't logged the link itself.
                                 */
                                        i_mac_perim_exit(mip);
                                        if (lstate->mi_last)
                                                return (0);
                                        else
                                                return (-1);
                                }
                                mcip->mci_state_flags |= MCIS_DESC_LOGGED;
                                list_insert_tail(lstate->mi_list, ninfo);
                        }
                }

                ninfo = mac_write_link_stats(mcip);
                if (ninfo == NULL && !lstate->mi_last) {
                        i_mac_perim_exit(mip);
                        return (-1);
                }
                list_insert_tail(lstate->mi_list, ninfo);

                if (lstate->mi_last)
                        mcip->mci_state_flags &= ~MCIS_DESC_LOGGED;

                if (lstate->mi_fenable) {
                        if (mcip->mci_subflow_tab != NULL) {
                                (void) mac_flow_walk_nolock(
                                    mcip->mci_subflow_tab, mac_log_flowinfo,
```

```
5479                                    lstate);
5480                        }
5481                }
5482        }
5483        i_mac_perim_exit(mip);
5484        return (0);
5485 }

5487 /*
5488  * modhash walker function to add a mac_impl_t to a list
5489  */
5490 /*ARGSUSED*/
5491 static uint_t
5492 i_mac_impl_list_walker(mod_hash_key_t key, mod_hash_val_t *val, void *arg)
5493 {
5494        list_t                  *list = (list_t *)arg;
5495        mac_impl_t              *mip = (mac_impl_t *)val;

5497        if ((mip->mi_state_flags & MIS_DISABLED) == 0) {
5498                list_insert_tail(list, mip);
5499                mip->mi_ref++;
5500        }

5502        return (MH_WALK_CONTINUE);
5503 }

5505 void
5506 i_mac_log_info(list_t *net_log_list, i_mac_log_state_t *lstate)
5507 {
5508        list_t                  mac_impl_list;
5509        mac_impl_t              *mip;
5510        netinfo_t               *ninfo;

5512        /* Create list of mac_impls */
5513        ASSERT(RW_LOCK_HELD(&i_mac_impl_lock));
5514        list_create(&mac_impl_list, sizeof (mac_impl_t), offsetof(mac_impl_t,
5515            mi_node));
5516        mod_hash_walk(i_mac_impl_hash, i_mac_impl_list_walker, &mac_impl_list);
5517        rw_exit(&i_mac_impl_lock);

5519        /* Create log entries for each mac_impl */
5520        for (mip = list_head(&mac_impl_list); mip != NULL;
5521            mip = list_next(&mac_impl_list, mip)) {
5522                if (i_mac_impl_log(mip, lstate) != 0)
5523                        continue;
5524        }

5526        /* Remove elements and destroy list of mac_impls */
5527        rw_enter(&i_mac_impl_lock, RW_WRITER);
5528        while ((mip = list_remove_tail(&mac_impl_list)) != NULL) {
5529                mip->mi_ref--;
5530        }
5531        rw_exit(&i_mac_impl_lock);
5532        list_destroy(&mac_impl_list);

5534        /*
5535         * Write log entries to files outside of locks, free associated
5536         * structures, and remove entries from the list.
5537         */
5538        while ((ninfo = list_head(net_log_list)) != NULL) {
5539                (void) exacct_commit_netinfo(ninfo->ni_record, ninfo->ni_type);
5540                list_remove(net_log_list, ninfo);
5541                kmem_free(ninfo->ni_record, ninfo->ni_size);
5542                kmem_free(ninfo, sizeof (*ninfo));
5543        }
5544        list_destroy(net_log_list);
```



```
5545 }

5547 /*
5548  * The timer thread that runs every mac_logging_interval seconds and logs
5549  * link and/or flow information.
5550  */
5551 /* ARGSUSED */
5552 void
5553 mac_log_linkinfo(void *arg)
5554 {
5555        i_mac_log_state_t       lstate;
5556        list_t                  net_log_list;

5558        list_create(&net_log_list, sizeof (netinfo_t),
5559            offsetof(netinfo_t, ni_link));

5561        rw_enter(&i_mac_impl_lock, RW_READER);
5562        if (!mac_flow_log_enable && !mac_link_log_enable) {
5563                rw_exit(&i_mac_impl_lock);
5564                return;
5565        }
5566        lstate.mi_fenable = mac_flow_log_enable;
5567        lstate.mi_lenable = mac_link_log_enable;
5568        lstate.mi_last = B_FALSE;
5569        lstate.mi_list = &net_log_list;

5571        /* Write log entries for each mac_impl in the list */
5572        i_mac_log_info(&net_log_list, &lstate);

5574        if (mac_flow_log_enable || mac_link_log_enable) {
5575                mac_logging_timer = timeout(mac_log_linkinfo, NULL,
5576                    SEC_TO_TICK(mac_logging_interval));
5577        }
5578 }

5580 typedef struct i_mac_fastpath_state_s {
5581        boolean_t       mf_disable;
5582        int             mf_err;
5583 } i_mac_fastpath_state_t;

5585 /* modhash walker function to enable or disable fastpath */
5586 /*ARGSUSED*/
5587 static uint_t
5588 i_mac_fastpath_walker(mod_hash_key_t key, mod_hash_val_t *val,
5589     void *arg)
5590 {
5591        i_mac_fastpath_state_t  *state = arg;
5592        mac_handle_t            mh = (mac_handle_t)val;

5594        if (state->mf_disable)
5595                state->mf_err = mac_fastpath_disable(mh);
5596        else
5597                mac_fastpath_enable(mh);

5599        return (state->mf_err == 0 ? MH_WALK_CONTINUE : MH_WALK_TERMINATE);
5600 }

5602 /*
5603  * Start the logging timer.
5604  */
5605 int
5606 mac_start_logusage(mac_logtype_t type, uint_t interval)
5607 {
5608        i_mac_fastpath_state_t  dstate = {B_TRUE, 0};
5609        i_mac_fastpath_state_t  estate = {B_FALSE, 0};
5610        int                     err;
```

```
5612         rw_enter(&i_mac_impl_lock, RW_WRITER);
5613         switch (type) {
5614         case MAC_LOGTYPE_FLOW:
5615                 if (mac_flow_log_enable) {
5616                         rw_exit(&i_mac_impl_lock);
5617                         return (0);
5618                 }
5619                 /* FALLTHRU */
5620         case MAC_LOGTYPE_LINK:
5621                 if (mac_link_log_enable) {
5622                         rw_exit(&i_mac_impl_lock);
5623                         return (0);
5624                 }
5625                 break;
5626         default:
5627                 ASSERT(0);
5628         }

5630         /* Disable fastpath */
5631         mod_hash_walk(i_mac_impl_hash, i_mac_fastpath_walker, &dstate);
5632         if ((err = dstate.mf_err) != 0) {
5633                 /* Reenable fastpath  */
5634                 mod_hash_walk(i_mac_impl_hash, i_mac_fastpath_walker, &estate);
5635                 rw_exit(&i_mac_impl_lock);
5636                 return (err);
5637         }

5639         switch (type) {
5640         case MAC_LOGTYPE_FLOW:
5641                 mac_flow_log_enable = B_TRUE;
5642                 /* FALLTHRU */
5643         case MAC_LOGTYPE_LINK:
5644                 mac_link_log_enable = B_TRUE;
5645                 break;
5646         }

5648         mac_logging_interval = interval;
5649         rw_exit(&i_mac_impl_lock);
5650         mac_log_linkinfo(NULL);
5651         return (0);
5652 }

5654 /*
5655  * Stop the logging timer if both link and flow logging are turned off.
5656  */
5657 void
5658 mac_stop_logusage(mac_logtype_t type)
5659 {
5660         i_mac_log_state_t       lstate;
5661         i_mac_fastpath_state_t  estate = {B_FALSE, 0};
5662         list_t                  net_log_list;

5664         list_create(&net_log_list, sizeof (netinfo_t),
5665             offsetof(netinfo_t, ni_link));

5667         rw_enter(&i_mac_impl_lock, RW_WRITER);

5669         lstate.mi_fenable = mac_flow_log_enable;
5670         lstate.mi_lenable = mac_link_log_enable;
5671         lstate.mi_list = &net_log_list;

5673         /* Last walk */
5674         lstate.mi_last = B_TRUE;

5676         switch (type) {
```



```
5677         case MAC_LOGTYPE_FLOW:
5678                 if (lstate.mi_fenable) {
5679                         ASSERT(mac_link_log_enable);
5680                         mac_flow_log_enable = B_FALSE;
5681                         mac_link_log_enable = B_FALSE;
5682                         break;
5683                 }
5684                 /* FALLTHRU */
5685         case MAC_LOGTYPE_LINK:
5686                 if (!lstate.mi_lenable || mac_flow_log_enable) {
5687                         rw_exit(&i_mac_impl_lock);
5688                         return;
5689                 }
5690                 mac_link_log_enable = B_FALSE;
5691                 break;
5692         default:
5693                 ASSERT(0);
5694         }

5696         /* Reenable fastpath */
5697         mod_hash_walk(i_mac_impl_hash, i_mac_fastpath_walker, &estate);

5699         (void) untimeout(mac_logging_timer);
5700         mac_logging_timer = 0;

5702         /* Write log entries for each mac_impl in the list */
5703         i_mac_log_info(&net_log_list, &lstate);
5704 }

5706 /*
5707  * Walk the rx and tx SRS/SRs for a flow and update the priority value.
5708  */
5709 void
5710 mac_flow_update_priority(mac_client_impl_t *mcip, flow_entry_t *flent)
5711 {
5712         pri_t                   pri;
5713         int                     count;
5714         mac_soft_ring_set_t     *mac_srs;

5716         if (flent->fe_rx_srs_cnt <= 0)
5717                 return;

5719         if (((mac_soft_ring_set_t *)flent->fe_rx_srs[0])->srs_type ==
5720             SRST_FLOW) {
5721                 pri = FLOW_PRIORITY(mcip->mci_min_pri,
5722                     mcip->mci_max_pri,
5723                     flent->fe_resource_props.mrp_priority);
5724         } else {
5725                 pri = mcip->mci_max_pri;
5726         }

5728         for (count = 0; count < flent->fe_rx_srs_cnt; count++) {
5729                 mac_srs = flent->fe_rx_srs[count];
5730                 mac_update_srs_priority(mac_srs, pri);
5731         }
5732         /*
5733          * If we have a Tx SRS, we need to modify all the threads associated
5734          * with it.
5735          */
5736         if (flent->fe_tx_srs != NULL)
5737                 mac_update_srs_priority(flent->fe_tx_srs, pri);
5738 }

5740 /*
5741  * RX and TX rings are reserved according to different semantics depending
5742  * on the requests from the MAC clients and type of rings:
```

```
5743  *
5744  * On the Tx side, by default we reserve individual rings, independently from
5745  * the groups.
5746  *
5747  * On the Rx side, the reservation is at the granularity of the group
5748  * of rings, and used for v12n level 1 only. It has a special case for the
5749  * primary client.
5750  *
5751  * If a share is allocated to a MAC client, we allocate a TX group and an
5752  * RX group to the client, and assign TX rings and RX rings to these
5753  * groups according to information gathered from the driver through
5754  * the share capability.
5755  *
5756  * The foreseable evolution of Rx rings will handle v12n level 2 and higher
5757  * to allocate individual rings out of a group and program the hw classifier
5758  * based on IP address or higher level criteria.
5759  */

5761 /*
5762  * mac_reserve_tx_ring()
5763  * Reserve a unused ring by marking it with MR_INUSE state.
5764  * As reserved, the ring is ready to function.
5765  *
5766  * Notes for Hybrid I/O:
5767  *
5768  * If a specific ring is needed, it is specified through the desired_ring
5769  * argument. Otherwise that argument is set to NULL.
5770  * If the desired ring was previous allocated to another client, this
5771  * function swaps it with a new ring from the group of unassigned rings.
5772  */
5773 mac_ring_t *
5774 mac_reserve_tx_ring(mac_impl_t *mip, mac_ring_t *desired_ring)
5775 {
5776         mac_group_t             *group;
5777         mac_grp_client_t        *mgcp;
5778         mac_client_impl_t       *mcip;
5779         mac_soft_ring_set_t     *srs;

5781         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

5783         /*
5784          * Find an available ring and start it before changing its status.
5785          * The unassigned rings are at the end of the mi_tx_groups
5786          * array.
5787          */
5788         group = MAC_DEFAULT_TX_GROUP(mip);

5790         /* Can't take the default ring out of the default group */
5791         ASSERT(desired_ring != (mac_ring_t *)mip->mi_default_tx_ring);

5793         if (desired_ring->mr_state == MR_FREE) {
5794                 ASSERT(MAC_GROUP_NO_CLIENT(group));
5795                 if (mac_start_ring(desired_ring) != 0)
5796                         return (NULL);
5797                 return (desired_ring);
5798         }
5799         /*
5800          * There are clients using this ring, so let's move the clients
5801          * away from using this ring.
5802          */
5803         for (mgcp = group->mrg_clients; mgcp != NULL; mgcp = mgcp->mgc_next) {
5804                 mcip = mgcp->mgc_client;
5805                 mac_tx_client_quiesce((mac_client_handle_t)mcip);
5806                 srs = MCIP_TX_SRS(mcip);
5807                 ASSERT(mac_tx_srs_ring_present(srs, desired_ring));
5808                 mac_tx_invoke_callbacks(mcip,
```



```
5809                     (mac_tx_cookie_t)mac_tx_srs_get_soft_ring(srs,
5810                     desired_ring));
5811                 mac_tx_srs_del_ring(srs, desired_ring);
5812                 mac_tx_client_restart((mac_client_handle_t)mcip);
5813         }
5814         return (desired_ring);
5815 }

5817 /*
5818  * For a reserved group with multiple clients, return the primary client.
5819  */
5820 static mac_client_impl_t *
5821 mac_get_grp_primary(mac_group_t *grp)
5822 {
5823         mac_grp_client_t        *mgcp = grp->mrg_clients;
5824         mac_client_impl_t       *mcip;

5826         while (mgcp != NULL) {
5827                 mcip = mgcp->mgc_client;
5828                 if (mcip->mci_flent->fe_type & FLOW_PRIMARY_MAC)
5829                         return (mcip);
5830                 mgcp = mgcp->mgc_next;
5831         }
5832         return (NULL);
5833 }

5835 /*
5836  * Hybrid I/O specifies the ring that should be given to a share.
5837  * If the ring is already used by clients, then we need to release
5838  * the ring back to the default group so that we can give it to
5839  * the share. This means the clients using this ring now get a
5840  * replacement ring. If there aren't any replacement rings, this
5841  * function returns a failure.
5842  */
5843 static int
5844 mac_reclaim_ring_from_grp(mac_impl_t *mip, mac_ring_type_t ring_type,
5845     mac_ring_t *ring, mac_ring_t **rings, int nrings)
5846 {
5847         mac_group_t             *group = (mac_group_t *)ring->mr_gh;
5848         mac_resource_props_t    *mrp;
5849         mac_client_impl_t       *mcip;
5850         mac_group_t             *defgrp;
5851         mac_ring_t              *tring;
5852         mac_group_t             *tgrp;
5853         int                     i;
5854         int                     j;

5856         mcip = MAC_GROUP_ONLY_CLIENT(group);
5857         if (mcip == NULL)
5858                 mcip = mac_get_grp_primary(group);
5859         ASSERT(mcip != NULL);
5860         ASSERT(mcip->mci_share == NULL);

5862         mrp = MCIP_RESOURCE_PROPS(mcip);
5863         if (ring_type == MAC_RING_TYPE_RX) {
5864                 defgrp = mip->mi_rx_donor_grp;
5865                 if ((mrp->mrp_mask & MRP_RX_RINGS) == 0) {
5866                         /* Need to put this mac client in the default group */
5867                         if (mac_rx_switch_group(mcip, group, defgrp) != 0)
5868                                 return (ENOSPC);
5869                 } else {
5870                         /*
5871                          * Switch this ring with some other ring from
5872                          * the default group.
5873                          */
5874                         for (tring = defgrp->mrg_rings; tring != NULL;
```

```
5875                            tring = tring->mr_next) {
5876                                if (tring->mr_index == 0)
5877                                        continue;
5878                                for (j = 0; j < nrings; j++) {
5879                                        if (rings[j] == tring)
5880                                                break;
5881                                }
5882                                if (j >= nrings)
5883                                        break;
5884                        }
5885                        if (tring == NULL)
5886                                return (ENOSPC);
5887                        if (mac_group_mov_ring(mip, group, tring) != 0)
5888                                return (ENOSPC);
5889                        if (mac_group_mov_ring(mip, defgrp, ring) != 0) {
5890                                (void) mac_group_mov_ring(mip, defgrp, tring);
5891                                return (ENOSPC);
5892                        }
5893                }
5894                ASSERT(ring->mr_gh == (mac_group_handle_t)defgrp);
5895                return (0);
5896        }

5898        defgrp = MAC_DEFAULT_TX_GROUP(mip);
5899        if (ring == (mac_ring_t *)mip->mi_default_tx_ring) {
5900                /*
5901                 * See if we can get a spare ring to replace the default
5902                 * ring.
5903                 */
5904                if (defgrp->mrg_cur_count == 1) {
5905                        /*
5906                         * Need to get a ring from another client, see if
5907                         * there are any clients that can be moved to
5908                         * the default group, thereby freeing some rings.
5909                         */
5910                        for (i = 0; i < mip->mi_tx_group_count; i++) {
5911                                tgrp = &mip->mi_tx_groups[i];
5912                                if (tgrp->mrg_state ==
5913                                    MAC_GROUP_STATE_REGISTERED) {
5914                                        continue;
5915                                }
5916                                mcip = MAC_GROUP_ONLY_CLIENT(tgrp);
5917                                if (mcip == NULL)
5918                                        mcip = mac_get_grp_primary(tgrp);
5919                                ASSERT(mcip != NULL);
5920                                mrp = MCIP_RESOURCE_PROPS(mcip);
5921                                if ((mrp->mrp_mask & MRP_TX_RINGS) == 0) {
5922                                        ASSERT(tgrp->mrg_cur_count == 1);
5923                                        /*
5924                                         * If this ring is part of the
5925                                         * rings asked by the share we cannot
5926                                         * use it as the default ring.
5927                                         */
5928                                        for (j = 0; j < nrings; j++) {
5929                                                if (rings[j] == tgrp->mrg_rings)
5930                                                        break;
5931                                        }
5932                                        if (j < nrings)
5933                                                continue;
5934                                        mac_tx_client_quiesce(
5935                                            (mac_client_handle_t)mcip);
5936                                        mac_tx_switch_group(mcip, tgrp,
5937                                            defgrp);
5938                                        mac_tx_client_restart(
5939                                            (mac_client_handle_t)mcip);
5940                                        break;
```



```
5941                                }
5942                        }
5943                        /*
5944                         * All the rings are reserved, can't give up the
5945                         * default ring.
5946                         */
5947                        if (defgrp->mrg_cur_count <= 1)
5948                                return (ENOSPC);
5949                }
5950                /*
5951                 * Swap the default ring with another.
5952                 */
5953                for (tring = defgrp->mrg_rings; tring != NULL;
5954                    tring = tring->mr_next) {
5955                        /*
5956                         * If this ring is part of the rings asked by the
5957                         * share we cannot use it as the default ring.
5958                         */
5959                        for (j = 0; j < nrings; j++) {
5960                                if (rings[j] == tring)
5961                                        break;
5962                        }
5963                        if (j >= nrings)
5964                                break;
5965                }
5966                ASSERT(tring != NULL);
5967                mip->mi_default_tx_ring = (mac_ring_handle_t)tring;
5968                return (0);
5969        }
5970        /*
5971         * The Tx ring is with a group reserved by a MAC client. See if
5972         * we can swap it.
5973         */
5974        ASSERT(group->mrg_state == MAC_GROUP_STATE_RESERVED);
5975        mcip = MAC_GROUP_ONLY_CLIENT(group);
5976        if (mcip == NULL)
5977                mcip = mac_get_grp_primary(group);
5978        ASSERT(mcip !=  NULL);
5979        mrp = MCIP_RESOURCE_PROPS(mcip);
5980        mac_tx_client_quiesce((mac_client_handle_t)mcip);
5981        if ((mrp->mrp_mask & MRP_TX_RINGS) == 0) {
5982                ASSERT(group->mrg_cur_count == 1);
5983                /* Put this mac client in the default group */
5984                mac_tx_switch_group(mcip, group, defgrp);
5985        } else {
5986                /*
5987                 * Switch this ring with some other ring from
5988                 * the default group.
5989                 */
5990                for (tring = defgrp->mrg_rings; tring != NULL;
5991                    tring = tring->mr_next) {
5992                        if (tring == (mac_ring_t *)mip->mi_default_tx_ring)
5993                                continue;
5994                        /*
5995                         * If this ring is part of the rings asked by the
5996                         * share we cannot use it for swapping.
5997                         */
5998                        for (j = 0; j < nrings; j++) {
5999                                if (rings[j] == tring)
6000                                        break;
6001                        }
6002                        if (j >= nrings)
6003                                break;
6004                }
6005                if (tring == NULL) {
6006                        mac_tx_client_restart((mac_client_handle_t)mcip);
```

```
6007                        return (ENOSPC);
6008                }
6009                if (mac_group_mov_ring(mip, group, tring) != 0) {
6010                        mac_tx_client_restart((mac_client_handle_t)mcip);
6011                        return (ENOSPC);
6012                }
6013                if (mac_group_mov_ring(mip, defgrp, ring) != 0) {
6014                        (void) mac_group_mov_ring(mip, defgrp, tring);
6015                        mac_tx_client_restart((mac_client_handle_t)mcip);
6016                        return (ENOSPC);
6017                }
6018        }
6019        mac_tx_client_restart((mac_client_handle_t)mcip);
6020        ASSERT(ring->mr_gh == (mac_group_handle_t)defgrp);
6021        return (0);
6022 }

6024 /*
6025  * Populate a zero-ring group with rings. If the share is non-NULL,
6026  * the rings are chosen according to that share.
6027  * Invoked after allocating a new RX or TX group through
6028  * mac_reserve_rx_group() or mac_reserve_tx_group(), respectively.
6029  * Returns zero on success, an errno otherwise.
6030  */
6031 int
6032 i_mac_group_allocate_rings(mac_impl_t *mip, mac_ring_type_t ring_type,
6033     mac_group_t *src_group, mac_group_t *new_group, mac_share_handle_t share,
6034     uint32_t ringcnt)
6035 {
6036        mac_ring_t **rings, *ring;
6037        uint_t nrings;
6038        int rv = 0, i = 0, j;

6040        ASSERT((ring_type == MAC_RING_TYPE_RX &&
6041            mip->mi_rx_group_type == MAC_GROUP_TYPE_DYNAMIC) ||
6042            (ring_type == MAC_RING_TYPE_TX &&
6043            mip->mi_tx_group_type == MAC_GROUP_TYPE_DYNAMIC));

6045        /*
6046         * First find the rings to allocate to the group.
6047         */
6048        if (share != NULL) {
6049                /* get rings through ms_squery() */
6050                mip->mi_share_capab.ms_squery(share, ring_type, NULL, &nrings);
6051                ASSERT(nrings != 0);
6052                rings = kmem_alloc(nrings * sizeof (mac_ring_handle_t),
6053                    KM_SLEEP);
6054                mip->mi_share_capab.ms_squery(share, ring_type,
6055                    (mac_ring_handle_t *)rings, &nrings);
6056                for (i = 0; i < nrings; i++) {
6057                        /*
6058                         * If we have given this ring to a non-default
6059                         * group, we need to check if we can get this
6060                         * ring.
6061                         */
6062                        ring = rings[i];
6063                        if (ring->mr_gh != (mac_group_handle_t)src_group ||
6064                            ring == (mac_ring_t *)mip->mi_default_tx_ring) {
6065                                if (mac_reclaim_ring_from_grp(mip, ring_type,
6066                                    ring, rings, nrings) != 0) {
6067                                        rv = ENOSPC;
6068                                        goto bail;
6069                                }
6070                        }
6071                }
6072        } else {
```



```
6073                /*
6074                 * Pick one ring from default group.
6075                 *
6076                 * for now pick the second ring which requires the first ring
6077                 * at index 0 to stay in the default group, since it is the
6078                 * ring which carries the multicast traffic.
6079                 * We need a better way for a driver to indicate this,
6080                 * for example a per-ring flag.
6081                 */
6082                rings = kmem_alloc(ringcnt * sizeof (mac_ring_handle_t),
6083                    KM_SLEEP);
6084                for (ring = src_group->mrg_rings; ring != NULL;
6085                    ring = ring->mr_next) {
6086                        if (ring_type == MAC_RING_TYPE_RX &&
6087                            ring->mr_index == 0) {
6088                                continue;
6089                        }
6090                        if (ring_type == MAC_RING_TYPE_TX &&
6091                            ring == (mac_ring_t *)mip->mi_default_tx_ring) {
6092                                continue;
6093                        }
6094                        rings[i++] = ring;
6095                        if (i == ringcnt)
6096                                break;
6097                }
6098                ASSERT(ring != NULL);
6099                nrings = i;
6100                /* Not enough rings as required */
6101                if (nrings != ringcnt) {
6102                        rv = ENOSPC;
6103                        goto bail;
6104                }
6105        }

6107        switch (ring_type) {
6108        case MAC_RING_TYPE_RX:
6109                if (src_group->mrg_cur_count - nrings < 1) {
6110                        /* we ran out of rings */
6111                        rv = ENOSPC;
6112                        goto bail;
6113                }

6115                /* move receive rings to new group */
6116                for (i = 0; i < nrings; i++) {
6117                        rv = mac_group_mov_ring(mip, new_group, rings[i]);
6118                        if (rv != 0) {
6119                                /* move rings back on failure */
6120                                for (j = 0; j < i; j++) {
6121                                        (void) mac_group_mov_ring(mip,
6122                                            src_group, rings[j]);
6123                                }
6124                                goto bail;
6125                        }
6126                }
6127                break;

6129        case MAC_RING_TYPE_TX: {
6130                mac_ring_t *tmp_ring;

6132                /* move the TX rings to the new group */
6133                for (i = 0; i < nrings; i++) {
6134                        /* get the desired ring */
6135                        tmp_ring = mac_reserve_tx_ring(mip, rings[i]);
6136                        if (tmp_ring == NULL) {
6137                                rv = ENOSPC;
6138                                goto bail;
```

```
6139                        }
6140                        ASSERT(tmp_ring == rings[i]);
6141                        rv = mac_group_mov_ring(mip, new_group, rings[i]);
6142                        if (rv != 0) {
6143                                /* cleanup on failure */
6144                                for (j = 0; j < i; j++) {
6145                                        (void) mac_group_mov_ring(mip,
6146                                            MAC_DEFAULT_TX_GROUP(mip),
6147                                            rings[j]);
6148                                }
6149                                goto bail;
6150                        }
6151                }
6152                break;
6153        }
6154        }

6156        /* add group to share */
6157        if (share != NULL)
6158                mip->mi_share_capab.ms_sadd(share, new_group->mrg_driver);

6160 bail:
6161        /* free temporary array of rings */
6162        kmem_free(rings, nrings * sizeof (mac_ring_handle_t));

6164        return (rv);
6165 }

6167 void
6168 mac_group_add_client(mac_group_t *grp, mac_client_impl_t *mcip)
6169 {
6170        mac_grp_client_t *mgcp;

6172        for (mgcp = grp->mrg_clients; mgcp != NULL; mgcp = mgcp->mgc_next) {
6173                if (mgcp->mgc_client == mcip)
6174                        break;
6175        }

6177        VERIFY(mgcp == NULL);

6179        mgcp = kmem_zalloc(sizeof (mac_grp_client_t), KM_SLEEP);
6180        mgcp->mgc_client = mcip;
6181        mgcp->mgc_next = grp->mrg_clients;
6182        grp->mrg_clients = mgcp;

6184 }

6186 void
6187 mac_group_remove_client(mac_group_t *grp, mac_client_impl_t *mcip)
6188 {
6189        mac_grp_client_t *mgcp, **pprev;

6191        for (pprev = &grp->mrg_clients, mgcp = *pprev; mgcp != NULL;
6192            pprev = &mgcp->mgc_next, mgcp = *pprev) {
6193                if (mgcp->mgc_client == mcip)
6194                        break;
6195        }

6197        ASSERT(mgcp != NULL);

6199        *pprev = mgcp->mgc_next;
6200        kmem_free(mgcp, sizeof (mac_grp_client_t));
6201 }

6203 /*
6204  * mac_reserve_rx_group()
```



```
6205  *
6206  * Finds an available group and exclusively reserves it for a client.
6207  * The group is chosen to suit the flow's resource controls (bandwidth and
6208  * fanout requirements) and the address type.
6209  * If the requestor is the pimary MAC then return the group with the
6210  * largest number of rings, otherwise the default ring when available.
6211  */
6212 mac_group_t *
6213 mac_reserve_rx_group(mac_client_impl_t *mcip, uint8_t *mac_addr, boolean_t move)
6214 {
6215        mac_share_handle_t      share = mcip->mci_share;
6216        mac_impl_t              *mip = mcip->mci_mip;
6217        mac_group_t             *grp = NULL;
6218        int                     i;
6219        int                     err = 0;
6220        mac_address_t           *map;
6221        mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);
6222        int                     nrings;
6223        int                     donor_grp_rcnt;
6224        boolean_t               need_exclgrp = B_FALSE;
6225        int                     need_rings = 0;
6226        mac_group_t             *candidate_grp = NULL;
6227        mac_client_impl_t       *gclient;
6228        mac_resource_props_t    *gmrp;
6229        mac_group_t             *donorgrp = NULL;
6230        boolean_t               rxhw = mrp->mrp_mask & MRP_RX_RINGS;
6231        boolean_t               unspec = mrp->mrp_mask & MRP_RXRINGS_UNSPEC;
6232        boolean_t               isprimary;

6234        ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

6236        isprimary = mcip->mci_flent->fe_type & FLOW_PRIMARY_MAC;

6238        /*
6239         * Check if a group already has this mac address (case of VLANs)
6240         * unless we are moving this MAC client from one group to another.
6241         */
6242        if (!move && (map = mac_find_macaddr(mip, mac_addr)) != NULL) {
6243                if (map->ma_group != NULL)
6244                        return (map->ma_group);
6245        }
6246        if (mip->mi_rx_groups == NULL || mip->mi_rx_group_count == 0)
6247                return (NULL);
6248        /*
6249         * If exclusive open, return NULL which will enable the
6250         * caller to use the default group.
6251         */
6252        if (mcip->mci_state_flags & MCIS_EXCLUSIVE)
6253                return (NULL);

6255        /* For dynamic groups default unspecified to 1 */
6256        if (rxhw && unspec &&
6257            mip->mi_rx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
6258                mrp->mrp_nrxrings = 1;
6259        }
6260        /*
6261         * For static grouping we allow only specifying rings=0 and
6262         * unspecified
6263         */
6264        if (rxhw && mrp->mrp_nrxrings > 0 &&
6265            mip->mi_rx_group_type == MAC_GROUP_TYPE_STATIC) {
6266                return (NULL);
6267        }
6268        if (rxhw) {
6269                /*
6270                 * We have explicitly asked for a group (with nrxrings,
```

```
6271                 * if unspec).
6272                 */
6273                if (unspec || mrp->mrp_nrxrings > 0) {
6274                        need_exclgrp = B_TRUE;
6275                        need_rings = mrp->mrp_nrxrings;
6276                } else if (mrp->mrp_nrxrings == 0) {
6277                        /*
6278                         * We have asked for a software group.
6279                         */
6280                        return (NULL);
6281                }
6282        } else if (isprimary && mip->mi_nactiveclients == 1 &&
6283            mip->mi_rx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
6284                /*
6285                 * If the primary is the only active client on this
6286                 * mip and we have not asked for any rings, we give
6287                 * it the default group so that the primary gets to
6288                 * use all the rings.
6289                 */
6290                return (NULL);
6291        }

6293        /* The group that can donate rings */
6294        donorgrp = mip->mi_rx_donor_grp;

6296        /*
6297         * The number of rings that the default group can donate.
6298         * We need to leave at least one ring.
6299         */
6300        donor_grp_rcnt = donorgrp->mrg_cur_count - 1;

6302        /*
6303         * Try to exclusively reserve a RX group.
6304         *
6305         * For flows requiring HW_DEFAULT_RING (unicast flow of the primary
6306         * client), try to reserve the a non-default RX group and give
6307         * it all the rings from the donor group, except the default ring
6308         *
6309         * For flows requiring HW_RING (unicast flow of other clients), try
6310         * to reserve non-default RX group with the specified number of
6311         * rings, if available.
6312         *
6313         * For flows that have not asked for software or hardware ring,
6314         * try to reserve a non-default group with 1 ring, if available.
6315         */
6316        for (i = 1; i < mip->mi_rx_group_count; i++) {
6317                grp = &mip->mi_rx_groups[i];

6319                DTRACE_PROBE3(rx__group__trying, char *, mip->mi_name,
6320                    int, grp->mrg_index, mac_group_state_t, grp->mrg_state);

6322                /*
6323                 * Check if this group could be a candidate group for
6324                 * eviction if we need a group for this MAC client,
6325                 * but there aren't any. A candidate group is one
6326                 * that didn't ask for an exclusive group, but got
6327                 * one and it has enough rings (combined with what
6328                 * the donor group can donate) for the new MAC
6329                 * client
6330                 */
6331                if (grp->mrg_state >= MAC_GROUP_STATE_RESERVED) {
6332                        /*
6333                         * If the primary/donor group is not the default
6334                         * group, don't bother looking for a candidate group.
6335                         * If we don't have enough rings we will check
6336                         * if the primary group can be vacated.
```



```
6337                         */
6338                        if (candidate_grp == NULL &&
6339                            donorgrp == MAC_DEFAULT_RX_GROUP(mip)) {
6340                                ASSERT(!MAC_GROUP_NO_CLIENT(grp));
6341                                gclient = MAC_GROUP_ONLY_CLIENT(grp);
6342                                if (gclient == NULL)
6343                                        gclient = mac_get_grp_primary(grp);
6344                                ASSERT(gclient != NULL);
6345                                gmrp = MCIP_RESOURCE_PROPS(gclient);
6346                                if (gclient->mci_share == NULL &&
6347                                    (gmrp->mrp_mask & MRP_RX_RINGS) == 0 &&
6348                                    (unspec ||
6349                                    (grp->mrg_cur_count + donor_grp_rcnt >=
6350                                    need_rings))) {
6351                                        candidate_grp = grp;
6352                                }
6353                        }
6354                        continue;
6355                }
6356                /*
6357                 * This group could already be SHARED by other multicast
6358                 * flows on this client. In that case, the group would
6359                 * be shared and has already been started.
6360                 */
6361                ASSERT(grp->mrg_state != MAC_GROUP_STATE_UNINIT);

6363                if ((grp->mrg_state == MAC_GROUP_STATE_REGISTERED) &&
6364                    (mac_start_group(grp) != 0)) {
6365                        continue;
6366                }

6368                if (mip->mi_rx_group_type != MAC_GROUP_TYPE_DYNAMIC)
6369                        break;
6370                ASSERT(grp->mrg_cur_count == 0);

6372                /*
6373                 * Populate the group. Rings should be taken
6374                 * from the donor group.
6375                 */
6376                nrings = rxhw ? need_rings : isprimary ? donor_grp_rcnt: 1;

6378                /*
6379                 * If the donor group can't donate, let's just walk and
6380                 * see if someone can vacate a group, so that we have
6381                 * enough rings for this, unless we already have
6382                 * identified a candiate group..
6383                 */
6384                if (nrings <= donor_grp_rcnt) {
6385                        err = i_mac_group_allocate_rings(mip, MAC_RING_TYPE_RX,
6386                            donorgrp, grp, share, nrings);
6387                        if (err == 0) {
6388                                /*
6389                                 * For a share i_mac_group_allocate_rings gets
6390                                 * the rings from the driver, let's populate
6391                                 * the property for the client now.
6392                                 */
6393                                if (share != NULL) {
6394                                        mac_client_set_rings(
6395                                            (mac_client_handle_t)mcip,
6396                                            grp->mrg_cur_count, -1);
6397                                }
6398                                if (mac_is_primary_client(mcip) && !rxhw)
6399                                        mip->mi_rx_donor_grp = grp;
6400                                break;
6401                        }
6402                }
```

```
6404                DTRACE_PROBE3(rx__group__reserve__alloc__rings, char *,
6405                    mip->mi_name, int, grp->mrg_index, int, err);

6407                /*
6408                 * It's a dynamic group but the grouping operation
6409                 * failed.
6410                 */
6411                mac_stop_group(grp);
6412        }
6413        /* We didn't find an exclusive group for this MAC client */
6414        if (i >= mip->mi_rx_group_count) {

6416                if (!need_exclgrp)
6417                        return (NULL);

6419                /*
6420                 * If we found a candidate group then we switch the
6421                 * MAC client from the candidate_group to the default
6422                 * group and give the group to this MAC client. If
6423                 * we didn't find a candidate_group, check if the
6424                 * primary is in its own group and if it can make way
6425                 * for this MAC client.
6426                 */
6427                if (candidate_grp == NULL &&
6428                    donorgrp != MAC_DEFAULT_RX_GROUP(mip) &&
6429                    donorgrp->mrg_cur_count >= need_rings) {
6430                        candidate_grp = donorgrp;
6431                }
6432                if (candidate_grp != NULL) {
6433                        boolean_t       prim_grp = B_FALSE;

6435                        /*
6436                         * Switch the MAC client from the candidate group
6437                         * to the default group.. If this group was the
6438                         * donor group, then after the switch we need
6439                         * to update the donor group too.
6440                         */
6441                        grp = candidate_grp;
6442                        gclient = MAC_GROUP_ONLY_CLIENT(grp);
6443                        if (gclient == NULL)
6444                                gclient = mac_get_grp_primary(grp);
6445                        if (grp == mip->mi_rx_donor_grp)
6446                                prim_grp = B_TRUE;
6447                        if (mac_rx_switch_group(gclient, grp,
6448                            MAC_DEFAULT_RX_GROUP(mip)) != 0) {
6449                                return (NULL);
6450                        }
6451                        if (prim_grp) {
6452                                mip->mi_rx_donor_grp =
6453                                    MAC_DEFAULT_RX_GROUP(mip);
6454                                donorgrp = MAC_DEFAULT_RX_GROUP(mip);
6455                        }


6458                        /*
6459                         * Now give this group with the required rings
6460                         * to this MAC client.
6461                         */
6462                        ASSERT(grp->mrg_state == MAC_GROUP_STATE_REGISTERED);
6463                        if (mac_start_group(grp) != 0)
6464                                return (NULL);

6466                        if (mip->mi_rx_group_type != MAC_GROUP_TYPE_DYNAMIC)
6467                                return (grp);
```



```
6469                        donor_grp_rcnt = donorgrp->mrg_cur_count - 1;
6470                        ASSERT(grp->mrg_cur_count == 0);
6471                        ASSERT(donor_grp_rcnt >= need_rings);
6472                        err = i_mac_group_allocate_rings(mip, MAC_RING_TYPE_RX,
6473                            donorgrp, grp, share, need_rings);
6474                        if (err == 0) {
6475                                /*
6476                                 * For a share i_mac_group_allocate_rings gets
6477                                 * the rings from the driver, let's populate
6478                                 * the property for the client now.
6479                                 */
6480                                if (share != NULL) {
6481                                        mac_client_set_rings(
6482                                            (mac_client_handle_t)mcip,
6483                                            grp->mrg_cur_count, -1);
6484                                }
6485                                DTRACE_PROBE2(rx__group__reserved,
6486                                    char *, mip->mi_name, int, grp->mrg_index);
6487                                return (grp);
6488                        }
6489                        DTRACE_PROBE3(rx__group__reserve__alloc__rings, char *,
6490                            mip->mi_name, int, grp->mrg_index, int, err);
6491                        mac_stop_group(grp);
6492                }
6493                return (NULL);
6494        }
6495        ASSERT(grp != NULL);

6497        DTRACE_PROBE2(rx__group__reserved,
6498            char *, mip->mi_name, int, grp->mrg_index);
6499        return (grp);
6500 }

6502 /*
6503  * mac_rx_release_group()
6504  *
6505  * This is called when there are no clients left for the group.
6506  * The group is stopped and marked MAC_GROUP_STATE_REGISTERED,
6507  * and if it is a non default group, the shares are removed and
6508  * all rings are assigned back to default group.
6509  */
6510 void
6511 mac_release_rx_group(mac_client_impl_t *mcip, mac_group_t *group)
6512 {
6513        mac_impl_t              *mip = mcip->mci_mip;
6514        mac_ring_t              *ring;

6516        ASSERT(group != MAC_DEFAULT_RX_GROUP(mip));

6518        if (mip->mi_rx_donor_grp == group)
6519                mip->mi_rx_donor_grp = MAC_DEFAULT_RX_GROUP(mip);

6521        /*
6522         * This is the case where there are no clients left. Any
6523         * SRS etc on this group have also be quiesced.
6524         */
6525        for (ring = group->mrg_rings; ring != NULL; ring = ring->mr_next) {
6526                if (ring->mr_classify_type == MAC_HW_CLASSIFIER) {
6527                        ASSERT(group->mrg_state == MAC_GROUP_STATE_RESERVED);
6528                        /*
6529                         * Remove the SRS associated with the HW ring.
6530                         * As a result, polling will be disabled.
6531                         */
6532                        ring->mr_srs = NULL;
6533                }
6534                ASSERT(group->mrg_state < MAC_GROUP_STATE_RESERVED ||
```

```
6535                    ring->mr_state == MR_INUSE);
6536                if (ring->mr_state == MR_INUSE) {
6537                        mac_stop_ring(ring);
6538                        ring->mr_flag = 0;
6539                }
6540        }

6542        /* remove group from share */
6543        if (mcip->mci_share != NULL) {
6544                mip->mi_share_capab.ms_sremove(mcip->mci_share,
6545                    group->mrg_driver);
6546        }

6548        if (mip->mi_rx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
6549                mac_ring_t *ring;

6551                /*
6552                 * Rings were dynamically allocated to group.
6553                 * Move rings back to default group.
6554                 */
6555                while ((ring = group->mrg_rings) != NULL) {
6556                        (void) mac_group_mov_ring(mip, mip->mi_rx_donor_grp,
6557                            ring);
6558                }
6559        }
6560        mac_stop_group(group);
6561        /*
6562         * Possible improvement: See if we can assign the group just released
6563         * to a another client of the mip
6564         */
6565 }

6567 /*
6568  * When we move the primary's mac address between groups, we need to also
6569  * take all the clients sharing the same mac address along with it (VLANs)
6570  * We remove the mac address for such clients from the group after quiescing
6571  * them. When we add the mac address we restart the client. Note that
6572  * the primary's mac address is removed from the group after all the
6573  * other clients sharing the address are removed. Similarly, the primary's
6574  * mac address is added before all the other client's mac address are
6575  * added. While grp is the group where the clients reside, tgrp is
6576  * the group where the addresses have to be added.
6577  */
6578 static void
6579 mac_rx_move_macaddr_prim(mac_client_impl_t *mcip, mac_group_t *grp,
6580     mac_group_t *tgrp, uint8_t *maddr, boolean_t add)
6581 {
6582        mac_impl_t              *mip = mcip->mci_mip;
6583        mac_grp_client_t        *mgcp = grp->mrg_clients;
6584        mac_client_impl_t       *gmcip;
6585        boolean_t               prim;

6587        prim = (mcip->mci_state_flags & MCIS_UNICAST_HW) != 0;

6589        /*
6590         * If the clients are in a non-default group, we just have to
6591         * walk the group's client list. If it is in the default group
6592         * (which will be shared by other clients as well, we need to
6593         * check if the unicast address matches mcip's unicast.
6594         */
6595        while (mgcp != NULL) {
6596                gmcip = mgcp->mgc_client;
6597                if (gmcip != mcip &&
6598                    (grp != MAC_DEFAULT_RX_GROUP(mip) ||
6599                    mcip->mci_unicast == gmcip->mci_unicast)) {
6600                        if (!add) {
```



```
6601                                mac_rx_client_quiesce(
6602                                    (mac_client_handle_t)gmcip);
6603                                (void) mac_remove_macaddr(mcip->mci_unicast);
6604                        } else {
6605                                (void) mac_add_macaddr(mip, tgrp, maddr, prim);
6606                                mac_rx_client_restart(
6607                                    (mac_client_handle_t)gmcip);
6608                        }
6609                }
6610                mgcp = mgcp->mgc_next;
6611        }
6612 }


6615 /*
6616  * Move the MAC address from fgrp to tgrp. If this is the primary client,
6617  * we need to take any VLANs etc. together too.
6618  */
6619 static int
6620 mac_rx_move_macaddr(mac_client_impl_t *mcip, mac_group_t *fgrp,
6621     mac_group_t *tgrp)
6622 {
6623        mac_impl_t              *mip = mcip->mci_mip;
6624        uint8_t                 maddr[MAXMACADDRLEN];
6625        int                     err = 0;
6626        boolean_t               prim;
6627        boolean_t               multiclnt = B_FALSE;

6629        mac_rx_client_quiesce((mac_client_handle_t)mcip);
6630        ASSERT(mcip->mci_unicast != NULL);
6631        bcopy(mcip->mci_unicast->ma_addr, maddr, mcip->mci_unicast->ma_len);

6633        prim = (mcip->mci_state_flags & MCIS_UNICAST_HW) != 0;
6634        if (mcip->mci_unicast->ma_nusers > 1) {
6635                mac_rx_move_macaddr_prim(mcip, fgrp, NULL, maddr, B_FALSE);
6636                multiclnt = B_TRUE;
6637        }
6638        ASSERT(mcip->mci_unicast->ma_nusers == 1);
6639        err = mac_remove_macaddr(mcip->mci_unicast);
6640        if (err != 0) {
6641                mac_rx_client_restart((mac_client_handle_t)mcip);
6642                if (multiclnt) {
6643                        mac_rx_move_macaddr_prim(mcip, fgrp, fgrp, maddr,
6644                            B_TRUE);
6645                }
6646                return (err);
6647        }
6648        /*
6649         * Program the H/W Classifier first, if this fails we need
6650         * not proceed with the other stuff.
6651         */
6652        if ((err = mac_add_macaddr(mip, tgrp, maddr, prim)) != 0) {
6653                /* Revert back the H/W Classifier */
6654                if ((err = mac_add_macaddr(mip, fgrp, maddr, prim)) != 0) {
6655                        /*
6656                         * This should not fail now since it worked earlier,
6657                         * should we panic?
6658                         */
6659                        cmn_err(CE_WARN,
6660                            "mac_rx_switch_group: switching %p back"
6661                            " to group %p failed!!", (void *)mcip,
6662                            (void *)fgrp);
6663                }
6664                mac_rx_client_restart((mac_client_handle_t)mcip);
6665                if (multiclnt) {
6666                        mac_rx_move_macaddr_prim(mcip, fgrp, fgrp, maddr,
```

```
6667                            B_TRUE);
6668                }
6669                return (err);
6670        }
6671        mcip->mci_unicast = mac_find_macaddr(mip, maddr);
6672        mac_rx_client_restart((mac_client_handle_t)mcip);
6673        if (multiclnt)
6674                mac_rx_move_macaddr_prim(mcip, fgrp, tgrp, maddr, B_TRUE);
6675        return (err);
6676 }

6678 /*
6679  * Switch the MAC client from one group to another. This means we need
6680  * to remove the MAC address from the group, remove the MAC client,
6681  * teardown the SRSs and revert the group state. Then, we add the client
6682  * to the destination group, set the SRSs, and add the MAC address to the
6683  * group.
6684  */
6685 int
6686 mac_rx_switch_group(mac_client_impl_t *mcip, mac_group_t *fgrp,
6687     mac_group_t *tgrp)
6688 {
6689        int                     err;
6690        mac_group_state_t       next_state;
6691        mac_client_impl_t       *group_only_mcip;
6692        mac_client_impl_t       *gmcip;
6693        mac_impl_t              *mip = mcip->mci_mip;
6694        mac_grp_client_t        *mgcp;

6696        ASSERT(fgrp == mcip->mci_flent->fe_rx_ring_group);

6698        if ((err = mac_rx_move_macaddr(mcip, fgrp, tgrp)) != 0)
6699                return (err);

6701        /*
6702         * The group might be reserved, but SRSs may not be set up, e.g.
6703         * primary and its vlans using a reserved group.
6704         */
6705        if (fgrp->mrg_state == MAC_GROUP_STATE_RESERVED &&
6706            MAC_GROUP_ONLY_CLIENT(fgrp) != NULL) {
6707                mac_rx_srs_group_teardown(mcip->mci_flent, B_TRUE);
6708        }
6709        if (fgrp != MAC_DEFAULT_RX_GROUP(mip)) {
6710                mgcp = fgrp->mrg_clients;
6711                while (mgcp != NULL) {
6712                        gmcip = mgcp->mgc_client;
6713                        mgcp = mgcp->mgc_next;
6714                        mac_group_remove_client(fgrp, gmcip);
6715                        mac_group_add_client(tgrp, gmcip);
6716                        gmcip->mci_flent->fe_rx_ring_group = tgrp;
6717                }
6718                mac_release_rx_group(mcip, fgrp);
6719                ASSERT(MAC_GROUP_NO_CLIENT(fgrp));
6720                mac_set_group_state(fgrp, MAC_GROUP_STATE_REGISTERED);
6721        } else {
6722                mac_group_remove_client(fgrp, mcip);
6723                mac_group_add_client(tgrp, mcip);
6724                mcip->mci_flent->fe_rx_ring_group = tgrp;
6725                /*
6726                 * If there are other clients (VLANs) sharing this address
6727                 * we should be here only for the primary.
6728                 */
6729                if (mcip->mci_unicast->ma_nusers > 1) {
6730                        /*
6731                         * We need to move all the clients that are using
6732                         * this h/w address.
```



```
6733                         */
6734                        mgcp = fgrp->mrg_clients;
6735                        while (mgcp != NULL) {
6736                                gmcip = mgcp->mgc_client;
6737                                mgcp = mgcp->mgc_next;
6738                                if (mcip->mci_unicast == gmcip->mci_unicast) {
6739                                        mac_group_remove_client(fgrp, gmcip);
6740                                        mac_group_add_client(tgrp, gmcip);
6741                                        gmcip->mci_flent->fe_rx_ring_group =
6742                                            tgrp;
6743                                }
6744                        }
6745                }
6746                /*
6747                 * The default group will still take the multicast,
6748                 * broadcast traffic etc., so it won't go to
6749                 * MAC_GROUP_STATE_REGISTERED.
6750                 */
6751                if (fgrp->mrg_state == MAC_GROUP_STATE_RESERVED)
6752                        mac_rx_group_unmark(fgrp, MR_CONDEMNED);
6753                mac_set_group_state(fgrp, MAC_GROUP_STATE_SHARED);
6754        }
6755        next_state = mac_group_next_state(tgrp, &group_only_mcip,
6756            MAC_DEFAULT_RX_GROUP(mip), B_TRUE);
6757        mac_set_group_state(tgrp, next_state);
6758        /*
6759         * If the destination group is reserved, setup the SRSs etc.
6760         */
6761        if (tgrp->mrg_state == MAC_GROUP_STATE_RESERVED) {
6762                mac_rx_srs_group_setup(mcip, mcip->mci_flent, SRST_LINK);
6763                mac_fanout_setup(mcip, mcip->mci_flent,
6764                    MCIP_RESOURCE_PROPS(mcip), mac_rx_deliver, mcip, NULL,
6765                    NULL);
6766                mac_rx_group_unmark(tgrp, MR_INCIPIENT);
6767        } else {
6768                mac_rx_switch_grp_to_sw(tgrp);
6769        }
6770        return (0);
6771 }

6773 /*
6774  * Reserves a TX group for the specified share. Invoked by mac_tx_srs_setup()
6775  * when a share was allocated to the client.
6776  */
6777 mac_group_t *
6778 mac_reserve_tx_group(mac_client_impl_t *mcip, boolean_t move)
6779 {
6780        mac_impl_t              *mip = mcip->mci_mip;
6781        mac_group_t             *grp = NULL;
6782        int                     rv;
6783        int                     i;
6784        int                     err;
6785        mac_group_t             *defgrp;
6786        mac_share_handle_t      share = mcip->mci_share;
6787        mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);
6788        int                     nrings;
6789        int                     defnrings;
6790        boolean_t               need_exclgrp = B_FALSE;
6791        int                     need_rings = 0;
6792        mac_group_t             *candidate_grp = NULL;
6793        mac_client_impl_t       *gclient;
6794        mac_resource_props_t    *gmrp;
6795        boolean_t               txhw = mrp->mrp_mask & MRP_TX_RINGS;
6796        boolean_t               unspec = mrp->mrp_mask & MRP_TXRINGS_UNSPEC;
6797        boolean_t               isprimary;
```

```
6799         isprimary = mcip->mci_flent->fe_type & FLOW_PRIMARY_MAC;
6800         /*
6801          * When we come here for a VLAN on the primary (dladm create-vlan),
6802          * we need to pair it along with the primary (to keep it consistent
6803          * with the RX side). So, we check if the primary is already assigned
6804          * to a group and return the group if so. The other way is also
6805          * true, i.e. the VLAN is already created and now we are plumbing
6806          * the primary.
6807          */
6808         if (!move && isprimary) {
6809                 for (gclient = mip->mi_clients_list; gclient != NULL;
6810                     gclient = gclient->mci_client_next) {
6811                         if (gclient->mci_flent->fe_type & FLOW_PRIMARY_MAC &&
6812                             gclient->mci_flent->fe_tx_ring_group != NULL) {
6813                                 return (gclient->mci_flent->fe_tx_ring_group);
6814                         }
6815                 }
6816         }

6818         if (mip->mi_tx_groups == NULL || mip->mi_tx_group_count == 0)
6819                 return (NULL);

6821         /* For dynamic groups, default unspec to 1 */
6822         if (txhw && unspec &&
6823             mip->mi_tx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
6824                 mrp->mrp_ntxrings = 1;
6825         }
6826         /*
6827          * For static grouping we allow only specifying rings=0 and
6828          * unspecified
6829          */
6830         if (txhw && mrp->mrp_ntxrings > 0 &&
6831             mip->mi_tx_group_type == MAC_GROUP_TYPE_STATIC) {
6832                 return (NULL);
6833         }

6835         if (txhw) {
6836                 /*
6837                  * We have explicitly asked for a group (with ntxrings,
6838                  * if unspec).
6839                  */
6840                 if (unspec || mrp->mrp_ntxrings > 0) {
6841                         need_exclgrp = B_TRUE;
6842                         need_rings = mrp->mrp_ntxrings;
6843                 } else if (mrp->mrp_ntxrings == 0) {
6844                         /*
6845                          * We have asked for a software group.
6846                          */
6847                         return (NULL);
6848                 }
6849         }
6850         defgrp = MAC_DEFAULT_TX_GROUP(mip);
6851         /*
6852          * The number of rings that the default group can donate.
6853          * We need to leave at least one ring - the default ring - in
6854          * this group.
6855          */
6856         defnrings = defgrp->mrg_cur_count - 1;

6858         /*
6859          * Primary gets default group unless explicitly told not
6860          * to  (i.e. rings > 0).
6861          */
6862         if (isprimary && !need_exclgrp)
6863                 return (NULL);
```



```
6865         nrings = (mrp->mrp_mask & MRP_TX_RINGS) != 0 ? mrp->mrp_ntxrings : 1;
6866         for (i = 0; i <  mip->mi_tx_group_count; i++) {
6867                 grp = &mip->mi_tx_groups[i];
6868                 if ((grp->mrg_state == MAC_GROUP_STATE_RESERVED) ||
6869                     (grp->mrg_state == MAC_GROUP_STATE_UNINIT)) {
6870                         /*
6871                          * Select a candidate for replacement if we don't
6872                          * get an exclusive group. A candidate group is one
6873                          * that didn't ask for an exclusive group, but got
6874                          * one and it has enough rings (combined with what
6875                          * the default group can donate) for the new MAC
6876                          * client.
6877                          */
6878                         if (grp->mrg_state == MAC_GROUP_STATE_RESERVED &&
6879                             candidate_grp == NULL) {
6880                                 gclient = MAC_GROUP_ONLY_CLIENT(grp);
6881                                 if (gclient == NULL)
6882                                         gclient = mac_get_grp_primary(grp);
6883                                 gmrp = MCIP_RESOURCE_PROPS(gclient);
6884                                 if (gclient->mci_share == NULL &&
6885                                     (gmrp->mrp_mask & MRP_TX_RINGS) == 0 &&
6886                                     (unspec ||
6887                                     (grp->mrg_cur_count + defnrings) >=
6888                                     need_rings)) {
6889                                         candidate_grp = grp;
6890                                 }
6891                         }
6892                         continue;
6893                 }
6894                 /*
6895                  * If the default can't donate let's just walk and
6896                  * see if someone can vacate a group, so that we have
6897                  * enough rings for this.
6898                  */
6899                 if (mip->mi_tx_group_type != MAC_GROUP_TYPE_DYNAMIC ||
6900                     nrings <= defnrings) {
6901                         if (grp->mrg_state == MAC_GROUP_STATE_REGISTERED) {
6902                                 rv = mac_start_group(grp);
6903                                 ASSERT(rv == 0);
6904                         }
6905                         break;
6906                 }
6907         }

6909         /* The default group */
6910         if (i >= mip->mi_tx_group_count) {
6911                 /*
6912                  * If we need an exclusive group and have identified a
6913                  * candidate group we switch the MAC client from the
6914                  * candidate group to the default group and give the
6915                  * candidate group to this client.
6916                  */
6917                 if (need_exclgrp && candidate_grp != NULL) {
6918                         /*
6919                          * Switch the MAC client from the candidate group
6920                          * to the default group.
6921                          */
6922                         grp = candidate_grp;
6923                         gclient = MAC_GROUP_ONLY_CLIENT(grp);
6924                         if (gclient == NULL)
6925                                 gclient = mac_get_grp_primary(grp);
6926                         mac_tx_client_quiesce((mac_client_handle_t)gclient);
6927                         mac_tx_switch_group(gclient, grp, defgrp);
6928                         mac_tx_client_restart((mac_client_handle_t)gclient);

6930                         /*
```

```
6931                         * Give the candidate group with the specified number
6932                         * of rings to this MAC client.
6933                         */
6934                        ASSERT(grp->mrg_state == MAC_GROUP_STATE_REGISTERED);
6935                        rv = mac_start_group(grp);
6936                        ASSERT(rv == 0);
6937
6938                        if (mip->mi_tx_group_type != MAC_GROUP_TYPE_DYNAMIC)
6939                                return (grp);
6940
6941                        ASSERT(grp->mrg_cur_count == 0);
6942                        ASSERT(defgrp->mrg_cur_count > need_rings);
6943
6944                        err = i_mac_group_allocate_rings(mip, MAC_RING_TYPE_TX,
6945                            defgrp, grp, share, need_rings);
6946                        if (err == 0) {
6947                                /*
6948                                 * For a share i_mac_group_allocate_rings gets
6949                                 * the rings from the driver, let's populate
6950                                 * the property for the client now.
6951                                 */
6952                                if (share != NULL) {
6953                                        mac_client_set_rings(
6954                                            (mac_client_handle_t)mcip, -1,
6955                                            grp->mrg_cur_count);
6956                                }
6957                                mip->mi_tx_group_free--;
6958                                return (grp);
6959                        }
6960                        DTRACE_PROBE3(tx__group__reserve__alloc__rings, char *,
6961                            mip->mi_name, int, grp->mrg_index, int, err);
6962                        mac_stop_group(grp);
6963                }
6964                return (NULL);
6965        }
6966        /*
6967         * We got an exclusive group, but it is not dynamic.
6968         */
6969        if (mip->mi_tx_group_type != MAC_GROUP_TYPE_DYNAMIC) {
6970                mip->mi_tx_group_free--;
6971                return (grp);
6972        }
6973
6974        rv = i_mac_group_allocate_rings(mip, MAC_RING_TYPE_TX, defgrp, grp,
6975            share, nrings);
6976        if (rv != 0) {
6977                DTRACE_PROBE3(tx__group__reserve__alloc__rings,
6978                    char *, mip->mi_name, int, grp->mrg_index, int, rv);
6979                mac_stop_group(grp);
6980                return (NULL);
6981        }
6982        /*
6983         * For a share i_mac_group_allocate_rings gets the rings from the
6984         * driver, let's populate the property for the client now.
6985         */
6986        if (share != NULL) {
6987                mac_client_set_rings((mac_client_handle_t)mcip, -1,
6988                    grp->mrg_cur_count);
6989        }
6990        mip->mi_tx_group_free--;
6991        return (grp);
6992 }
6993
6994 void
6995 mac_release_tx_group(mac_client_impl_t *mcip, mac_group_t *grp)
6996 {
```



```
6997        mac_impl_t              *mip = mcip->mci_mip;
6998        mac_share_handle_t      share = mcip->mci_share;
6999        mac_ring_t              *ring;
7000        mac_soft_ring_set_t     *srs = MCIP_TX_SRS(mcip);
7001        mac_group_t             *defgrp;
7002
7003        defgrp = MAC_DEFAULT_TX_GROUP(mip);
7004        if (srs != NULL) {
7005                if (srs->srs_soft_ring_count > 0) {
7006                        for (ring = grp->mrg_rings; ring != NULL;
7007                            ring = ring->mr_next) {
7008                                ASSERT(mac_tx_srs_ring_present(srs, ring));
7009                                mac_tx_invoke_callbacks(mcip,
7010                                    (mac_tx_cookie_t)
7011                                    mac_tx_srs_get_soft_ring(srs, ring));
7012                                mac_tx_srs_del_ring(srs, ring);
7013                        }
7014                } else {
7015                        ASSERT(srs->srs_tx.st_arg2 != NULL);
7016                        srs->srs_tx.st_arg2 = NULL;
7017                        mac_srs_stat_delete(srs);
7018                }
7019        }
7020        if (share != NULL)
7021                mip->mi_share_capab.ms_sremove(share, grp->mrg_driver);
7022
7023        /* move the ring back to the pool */
7024        if (mip->mi_tx_group_type == MAC_GROUP_TYPE_DYNAMIC) {
7025                while ((ring = grp->mrg_rings) != NULL)
7026                        (void) mac_group_mov_ring(mip, defgrp, ring);
7027        }
7028        mac_stop_group(grp);
7029        mip->mi_tx_group_free++;
7030 }
7031
7032 /*
7033  * Disassociate a MAC client from a group, i.e go through the rings in the
7034  * group and delete all the soft rings tied to them.
7035  */
7036 static void
7037 mac_tx_dismantle_soft_rings(mac_group_t *fgrp, flow_entry_t *flent)
7038 {
7039        mac_client_impl_t       *mcip = flent->fe_mcip;
7040        mac_soft_ring_set_t     *tx_srs;
7041        mac_srs_tx_t            *tx;
7042        mac_ring_t              *ring;
7043
7044        tx_srs = flent->fe_tx_srs;
7045        tx = &tx_srs->srs_tx;
7046
7047        /* Single ring case we haven't created any soft rings */
7048        if (tx->st_mode == SRS_TX_BW || tx->st_mode == SRS_TX_SERIALIZE ||
7049            tx->st_mode == SRS_TX_DEFAULT) {
7050                tx->st_arg2 = NULL;
7051                mac_srs_stat_delete(tx_srs);
7052        /* Fanout case, where we have to dismantle the soft rings */
7053        } else {
7054                for (ring = fgrp->mrg_rings; ring != NULL;
7055                    ring = ring->mr_next) {
7056                        ASSERT(mac_tx_srs_ring_present(tx_srs, ring));
7057                        mac_tx_invoke_callbacks(mcip,
7058                            (mac_tx_cookie_t)mac_tx_srs_get_soft_ring(tx_srs,
7059                            ring));
7060                        mac_tx_srs_del_ring(tx_srs, ring);
7061                }
7062                ASSERT(tx->st_arg2 == NULL);
```

```
7063         }
7064 }

7066 /*
7067  * Switch the MAC client from one group to another. This means we need
7068  * to remove the MAC client, teardown the SRSs and revert the group state.
7069  * Then, we add the client to the destination roup, set the SRSs etc.
7070  */
7071 void
7072 mac_tx_switch_group(mac_client_impl_t *mcip, mac_group_t *fgrp,
7073     mac_group_t *tgrp)
7074 {
7075         mac_client_impl_t       *group_only_mcip;
7076         mac_impl_t              *mip = mcip->mci_mip;
7077         flow_entry_t            *flent = mcip->mci_flent;
7078         mac_group_t             *defgrp;
7079         mac_grp_client_t        *mgcp;
7080         mac_client_impl_t       *gmcip;
7081         flow_entry_t            *gflent;

7083         defgrp = MAC_DEFAULT_TX_GROUP(mip);
7084         ASSERT(fgrp == flent->fe_tx_ring_group);

7086         if (fgrp == defgrp) {
7087                 /*
7088                  * If this is the primary we need to find any VLANs on
7089                  * the primary and move them too.
7090                  */
7091                 mac_group_remove_client(fgrp, mcip);
7092                 mac_tx_dismantle_soft_rings(fgrp, flent);
7093                 if (mcip->mci_unicast->ma_nusers > 1) {
7094                         mgcp = fgrp->mrg_clients;
7095                         while (mgcp != NULL) {
7096                                 gmcip = mgcp->mgc_client;
7097                                 mgcp = mgcp->mgc_next;
7098                                 if (mcip->mci_unicast != gmcip->mci_unicast)
7099                                         continue;
7100                                 mac_tx_client_quiesce(
7101                                     (mac_client_handle_t)gmcip);

7103                                 gflent = gmcip->mci_flent;
7104                                 mac_group_remove_client(fgrp, gmcip);
7105                                 mac_tx_dismantle_soft_rings(fgrp, gflent);

7107                                 mac_group_add_client(tgrp, gmcip);
7108                                 gflent->fe_tx_ring_group = tgrp;
7109                                 /* We could directly set this to SHARED */
7110                                 tgrp->mrg_state = mac_group_next_state(tgrp,
7111                                     &group_only_mcip, defgrp, B_FALSE);

7113                                 mac_tx_srs_group_setup(gmcip, gflent,
7114                                     SRST_LINK);
7115                                 mac_fanout_setup(gmcip, gflent,
7116                                     MCIP_RESOURCE_PROPS(gmcip), mac_rx_deliver,
7117                                     gmcip, NULL, NULL);

7119                                 mac_tx_client_restart(
7120                                     (mac_client_handle_t)gmcip);
7121                         }
7122                 }
7123                 if (MAC_GROUP_NO_CLIENT(fgrp)) {
7124                         mac_ring_t      *ring;
7125                         int             cnt;
7126                         int             ringcnt;

7128                         fgrp->mrg_state = MAC_GROUP_STATE_REGISTERED;
```



```
7129                         /*
7130                          * Additionally, we also need to stop all
7131                          * the rings in the default group, except
7132                          * the default ring. The reason being
7133                          * this group won't be released since it is
7134                          * the default group, so the rings won't
7135                          * be stopped otherwise.
7136                          */
7137                         ringcnt = fgrp->mrg_cur_count;
7138                         ring = fgrp->mrg_rings;
7139                         for (cnt = 0; cnt < ringcnt; cnt++) {
7140                                 if (ring->mr_state == MR_INUSE &&
7141                                     ring !=
7142                                     (mac_ring_t *)mip->mi_default_tx_ring) {
7143                                         mac_stop_ring(ring);
7144                                         ring->mr_flag = 0;
7145                                 }
7146                                 ring = ring->mr_next;
7147                         }
7148                 } else if (MAC_GROUP_ONLY_CLIENT(fgrp) != NULL) {
7149                         fgrp->mrg_state = MAC_GROUP_STATE_RESERVED;
7150                 } else {
7151                         ASSERT(fgrp->mrg_state == MAC_GROUP_STATE_SHARED);
7152                 }
7153         } else {
7154                 /*
7155                  * We could have VLANs sharing the non-default group with
7156                  * the primary.
7157                  */
7158                 mgcp = fgrp->mrg_clients;
7159                 while (mgcp != NULL) {
7160                         gmcip = mgcp->mgc_client;
7161                         mgcp = mgcp->mgc_next;
7162                         if (gmcip == mcip)
7163                                 continue;
7164                         mac_tx_client_quiesce((mac_client_handle_t)gmcip);
7165                         gflent = gmcip->mci_flent;

7167                         mac_group_remove_client(fgrp, gmcip);
7168                         mac_tx_dismantle_soft_rings(fgrp, gflent);

7170                         mac_group_add_client(tgrp, gmcip);
7171                         gflent->fe_tx_ring_group = tgrp;
7172                         /* We could directly set this to SHARED */
7173                         tgrp->mrg_state = mac_group_next_state(tgrp,
7174                             &group_only_mcip, defgrp, B_FALSE);
7175                         mac_tx_srs_group_setup(gmcip, gflent, SRST_LINK);
7176                         mac_fanout_setup(gmcip, gflent,
7177                             MCIP_RESOURCE_PROPS(gmcip), mac_rx_deliver,
7178                             gmcip, NULL, NULL);

7180                         mac_tx_client_restart((mac_client_handle_t)gmcip);
7181                 }
7182                 mac_group_remove_client(fgrp, mcip);
7183                 mac_release_tx_group(mcip, fgrp);
7184                 fgrp->mrg_state = MAC_GROUP_STATE_REGISTERED;
7185         }

7187         /* Add it to the tgroup */
7188         mac_group_add_client(tgrp, mcip);
7189         flent->fe_tx_ring_group = tgrp;
7190         tgrp->mrg_state = mac_group_next_state(tgrp, &group_only_mcip,
7191             defgrp, B_FALSE);

7193         mac_tx_srs_group_setup(mcip, flent, SRST_LINK);
7194         mac_fanout_setup(mcip, flent, MCIP_RESOURCE_PROPS(mcip),
```

```
7195             mac_rx_deliver, mcip, NULL, NULL);
7196 }

7198 /*
7199  * This is a 1-time control path activity initiated by the client (IP).
7200  * The mac perimeter protects against other simultaneous control activities,
7201  * for example an ioctl that attempts to change the degree of fanout and
7202  * increase or decrease the number of softrings associated with this Tx SRS.
7203  */
7204 static mac_tx_notify_cb_t *
7205 mac_client_tx_notify_add(mac_client_impl_t *mcip,
7206     mac_tx_notify_t notify, void *arg)
7207 {
7208         mac_cb_info_t *mcbi;
7209         mac_tx_notify_cb_t *mtnfp;

7211         ASSERT(MAC_PERIM_HELD((mac_handle_t)mcip->mci_mip));

7213         mtnfp = kmem_zalloc(sizeof (mac_tx_notify_cb_t), KM_SLEEP);
7214         mtnfp->mtnf_fn = notify;
7215         mtnfp->mtnf_arg = arg;
7216         mtnfp->mtnf_link.mcb_objp = mtnfp;
7217         mtnfp->mtnf_link.mcb_objsize = sizeof (mac_tx_notify_cb_t);
7218         mtnfp->mtnf_link.mcb_flags = MCB_TX_NOTIFY_CB_T;

7220         mcbi = &mcip->mci_tx_notify_cb_info;
7221         mutex_enter(mcbi->mcbi_lockp);
7222         mac_callback_add(mcbi, &mcip->mci_tx_notify_cb_list, &mtnfp->mtnf_link);
7223         mutex_exit(mcbi->mcbi_lockp);
7224         return (mtnfp);
7225 }

7227 static void
7228 mac_client_tx_notify_remove(mac_client_impl_t *mcip, mac_tx_notify_cb_t *mtnfp)
7229 {
7230         mac_cb_info_t   *mcbi;
7231         mac_cb_t        **cblist;

7233         ASSERT(MAC_PERIM_HELD((mac_handle_t)mcip->mci_mip));

7235         if (!mac_callback_find(&mcip->mci_tx_notify_cb_info,
7236             &mcip->mci_tx_notify_cb_list, &mtnfp->mtnf_link)) {
7237                 cmn_err(CE_WARN,
7238                     "mac_client_tx_notify_remove: callback not "
7239                     "found, mcip 0x%p mtnfp 0x%p", (void *)mcip, (void *)mtnfp);
7240                 return;
7241         }

7243         mcbi = &mcip->mci_tx_notify_cb_info;
7244         cblist = &mcip->mci_tx_notify_cb_list;
7245         mutex_enter(mcbi->mcbi_lockp);
7246         if (mac_callback_remove(mcbi, cblist, &mtnfp->mtnf_link))
7247                 kmem_free(mtnfp, sizeof (mac_tx_notify_cb_t));
7248         else
7249                 mac_callback_remove_wait(&mcip->mci_tx_notify_cb_info);
7250         mutex_exit(mcbi->mcbi_lockp);
7251 }

7253 /*
7254  * mac_client_tx_notify():
7255  * call to add and remove flow control callback routine.
7256  */
7257 mac_tx_notify_handle_t
7258 mac_client_tx_notify(mac_client_handle_t mch, mac_tx_notify_t callb_func,
7259     void *ptr)
7260 {
```



```
7261         mac_client_impl_t       *mcip = (mac_client_impl_t *)mch;
7262         mac_tx_notify_cb_t      *mtnfp = NULL;

7264         i_mac_perim_enter(mcip->mci_mip);

7266         if (callb_func != NULL) {
7267                 /* Add a notify callback */
7268                 mtnfp = mac_client_tx_notify_add(mcip, callb_func, ptr);
7269         } else {
7270                 mac_client_tx_notify_remove(mcip, (mac_tx_notify_cb_t *)ptr);
7271         }
7272         i_mac_perim_exit(mcip->mci_mip);

7274         return ((mac_tx_notify_handle_t)mtnfp);
7275 }

7277 void
7278 mac_bridge_vectors(mac_bridge_tx_t txf, mac_bridge_rx_t rxf,
7279     mac_bridge_ref_t reff, mac_bridge_ls_t lsf)
7280 {
7281         mac_bridge_tx_cb = txf;
7282         mac_bridge_rx_cb = rxf;
7283         mac_bridge_ref_cb = reff;
7284         mac_bridge_ls_cb = lsf;
7285 }

7287 int
7288 mac_bridge_set(mac_handle_t mh, mac_handle_t link)
7289 {
7290         mac_impl_t *mip = (mac_impl_t *)mh;
7291         int retv;

7293         mutex_enter(&mip->mi_bridge_lock);
7294         if (mip->mi_bridge_link == NULL) {
7295                 mip->mi_bridge_link = link;
7296                 retv = 0;
7297         } else {
7298                 retv = EBUSY;
7299         }
7300         mutex_exit(&mip->mi_bridge_lock);
7301         if (retv == 0) {
7302                 mac_poll_state_change(mh, B_FALSE);
7303                 mac_capab_update(mh);
7304         }
7305         return (retv);
7306 }

7308 /*
7309  * Disable bridging on the indicated link.
7310  */
7311 void
7312 mac_bridge_clear(mac_handle_t mh, mac_handle_t link)
7313 {
7314         mac_impl_t *mip = (mac_impl_t *)mh;

7316         mutex_enter(&mip->mi_bridge_lock);
7317         ASSERT(mip->mi_bridge_link == link);
7318         mip->mi_bridge_link = NULL;
7319         mutex_exit(&mip->mi_bridge_lock);
7320         mac_poll_state_change(mh, B_TRUE);
7321         mac_capab_update(mh);
7322 }

7324 void
7325 mac_no_active(mac_handle_t mh)
7326 {
```

```
7327         mac_impl_t *mip = (mac_impl_t *)mh;

7329         i_mac_perim_enter(mip);
7330         mip->mi_state_flags |= MIS_NO_ACTIVE;
7331         i_mac_perim_exit(mip);
7332 }

7334 /*
7335  * Walk the primary VLAN clients whenever the primary's rings property
7336  * changes and update the mac_resource_props_t for the VLAN's client.
7337  * We need to do this since we don't support setting these properties
7338  * on the primary's VLAN clients, but the VLAN clients have to
7339  * follow the primary w.r.t the rings property;
7340  */
7341 void
7342 mac_set_prim_vlan_rings(mac_impl_t  *mip, mac_resource_props_t *mrp)
7343 {
7344         mac_client_impl_t       *vmcip;
7345         mac_resource_props_t    *vmrp;

7347         for (vmcip = mip->mi_clients_list; vmcip != NULL;
7348             vmcip = vmcip->mci_client_next) {
7349                 if (!(vmcip->mci_flent->fe_type & FLOW_PRIMARY_MAC) ||
7350                     mac_client_vid((mac_client_handle_t)vmcip) ==
7351                     VLAN_ID_NONE) {
7352                         continue;
7353                 }
7354                 vmrp = MCIP_RESOURCE_PROPS(vmcip);

7356                 vmrp->mrp_nrxrings =  mrp->mrp_nrxrings;
7357                 if (mrp->mrp_mask & MRP_RX_RINGS)
7358                         vmrp->mrp_mask |= MRP_RX_RINGS;
7359                 else if (vmrp->mrp_mask & MRP_RX_RINGS)
7360                         vmrp->mrp_mask &= ~MRP_RX_RINGS;

7362                 vmrp->mrp_ntxrings =  mrp->mrp_ntxrings;
7363                 if (mrp->mrp_mask & MRP_TX_RINGS)
7364                         vmrp->mrp_mask |= MRP_TX_RINGS;
7365                 else if (vmrp->mrp_mask & MRP_TX_RINGS)
7366                         vmrp->mrp_mask &= ~MRP_TX_RINGS;

7368                 if (mrp->mrp_mask & MRP_RXRINGS_UNSPEC)
7369                         vmrp->mrp_mask |= MRP_RXRINGS_UNSPEC;
7370                 else
7371                         vmrp->mrp_mask &= ~MRP_RXRINGS_UNSPEC;

7373                 if (mrp->mrp_mask & MRP_TXRINGS_UNSPEC)
7374                         vmrp->mrp_mask |= MRP_TXRINGS_UNSPEC;
7375                 else
7376                         vmrp->mrp_mask &= ~MRP_TXRINGS_UNSPEC;
7377         }
7378 }

7380 /*
7381  * We are adding or removing ring(s) from a group. The source for taking
7382  * rings is the default group. The destination for giving rings back is
7383  * the default group.
7384  */
7385 int
7386 mac_group_ring_modify(mac_client_impl_t *mcip, mac_group_t *group,
7387     mac_group_t *defgrp)
7388 {
7389         mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);
7390         uint_t                  modify;
7391         int                     count;
7392         mac_ring_t              *ring;
```



```
7393         mac_ring_t              *next;
7394         mac_impl_t              *mip = mcip->mci_mip;
7395         mac_ring_t              **rings;
7396         uint_t                  ringcnt;
7397         int                     i = 0;
7398         boolean_t               rx_group = group->mrg_type == MAC_RING_TYPE_RX;
7399         int                     start;
7400         int                     end;
7401         mac_group_t             *tgrp;
7402         int                     j;
7403         int                     rv = 0;

7405         /*
7406          * If we are asked for just a group, we give 1 ring, else
7407          * the specified number of rings.
7408          */
7409         if (rx_group) {
7410                 ringcnt = (mrp->mrp_mask & MRP_RXRINGS_UNSPEC) ? 1:
7411                     mrp->mrp_nrxrings;
7412         } else {
7413                 ringcnt = (mrp->mrp_mask & MRP_TXRINGS_UNSPEC) ? 1:
7414                     mrp->mrp_ntxrings;
7415         }

7417         /* don't allow modifying rings for a share for now. */
7418         ASSERT(mcip->mci_share == NULL);

7420         if (ringcnt == group->mrg_cur_count)
7421                 return (0);

7423         if (group->mrg_cur_count > ringcnt) {
7424                 modify = group->mrg_cur_count - ringcnt;
7425                 if (rx_group) {
7426                         if (mip->mi_rx_donor_grp == group) {
7427                                 ASSERT(mac_is_primary_client(mcip));
7428                                 mip->mi_rx_donor_grp = defgrp;
7429                         } else {
7430                                 defgrp = mip->mi_rx_donor_grp;
7431                         }
7432                 }
7433                 ring = group->mrg_rings;
7434                 rings = kmem_alloc(modify * sizeof (mac_ring_handle_t),
7435                     KM_SLEEP);
7436                 j = 0;
7437                 for (count = 0; count < modify; count++) {
7438                         next = ring->mr_next;
7439                         rv = mac_group_mov_ring(mip, defgrp, ring);
7440                         if (rv != 0) {
7441                                 /* cleanup on failure */
7442                                 for (j = 0; j < count; j++) {
7443                                         (void) mac_group_mov_ring(mip, group,
7444                                             rings[j]);
7445                                 }
7446                                 break;
7447                         }
7448                         rings[j++] = ring;
7449                         ring = next;
7450                 }
7451                 kmem_free(rings, modify * sizeof (mac_ring_handle_t));
7452                 return (rv);
7453         }
7454         if (ringcnt >= MAX_RINGS_PER_GROUP)
7455                 return (EINVAL);

7457         modify = ringcnt - group->mrg_cur_count;
```

```
7459         if (rx_group) {
7460                 if (group != mip->mi_rx_donor_grp)
7461                         defgrp = mip->mi_rx_donor_grp;
7462                 else
7463                         /*
7464                          * This is the donor group with all the remaining
7465                          * rings. Default group now gets to be the donor
7466                          */
7467                         mip->mi_rx_donor_grp = defgrp;
7468                 start = 1;
7469                 end = mip->mi_rx_group_count;
7470         } else {
7471                 start = 0;
7472                 end = mip->mi_tx_group_count - 1;
7473         }
7474         /*
7475          * If the default doesn't have any rings, lets see if we can
7476          * take rings given to an h/w client that doesn't need it.
7477          * For now, we just see if there is  any one client that can donate
7478          * all the required rings.
7479          */
7480         if (defgrp->mrg_cur_count < (modify + 1)) {
7481                 for (i = start; i < end; i++) {
7482                         if (rx_group) {
7483                                 tgrp = &mip->mi_rx_groups[i];
7484                                 if (tgrp == group || tgrp->mrg_state <
7485                                     MAC_GROUP_STATE_RESERVED) {
7486                                         continue;
7487                                 }
7488                                 mcip = MAC_GROUP_ONLY_CLIENT(tgrp);
7489                                 if (mcip == NULL)
7490                                         mcip = mac_get_grp_primary(tgrp);
7491                                 ASSERT(mcip != NULL);
7492                                 mrp = MCIP_RESOURCE_PROPS(mcip);
7493                                 if ((mrp->mrp_mask & MRP_RX_RINGS) != 0)
7494                                         continue;
7495                                 if ((tgrp->mrg_cur_count +
7496                                     defgrp->mrg_cur_count) < (modify + 1)) {
7497                                         continue;
7498                                 }
7499                                 if (mac_rx_switch_group(mcip, tgrp,
7500                                     defgrp) != 0) {
7501                                         return (ENOSPC);
7502                                 }
7503                         } else {
7504                                 tgrp = &mip->mi_tx_groups[i];
7505                                 if (tgrp == group || tgrp->mrg_state <
7506                                     MAC_GROUP_STATE_RESERVED) {
7507                                         continue;
7508                                 }
7509                                 mcip = MAC_GROUP_ONLY_CLIENT(tgrp);
7510                                 if (mcip == NULL)
7511                                         mcip = mac_get_grp_primary(tgrp);
7512                                 mrp = MCIP_RESOURCE_PROPS(mcip);
7513                                 if ((mrp->mrp_mask & MRP_TX_RINGS) != 0)
7514                                         continue;
7515                                 if ((tgrp->mrg_cur_count +
7516                                     defgrp->mrg_cur_count) < (modify + 1)) {
7517                                         continue;
7518                                 }
7519                                 /* OK, we can switch this to s/w */
7520                                 mac_tx_client_quiesce(
7521                                     (mac_client_handle_t)mcip);
7522                                 mac_tx_switch_group(mcip, tgrp, defgrp);
7523                                 mac_tx_client_restart(
7524                                     (mac_client_handle_t)mcip);
```



```
7525                         }
7526                 }
7527                 if (defgrp->mrg_cur_count < (modify + 1))
7528                         return (ENOSPC);
7529         }
7530         if ((rv = i_mac_group_allocate_rings(mip, group->mrg_type, defgrp,
7531             group, mcip->mci_share, modify)) != 0) {
7532                 return (rv);
7533         }
7534         return (0);
7535 }

7537 /*
7538  * Given the poolname in mac_resource_props, find the cpupart
7539  * that is associated with this pool.  The cpupart will be used
7540  * later for finding the cpus to be bound to the networking threads.
7541  *
7542  * use_default is set B_TRUE if pools are enabled and pool_default
7543  * is returned.  This avoids a 2nd lookup to set the poolname
7544  * for pool-effective.
7545  *
7546  * returns:
7547  *
7548  *    NULL -   pools are disabled or if the 'cpus' property is set.
7549  *    cpupart of pool_default  - pools are enabled and the pool
7550  *             is not available or poolname is blank
7551  *    cpupart of named pool    - pools are enabled and the pool
7552  *             is available.
7553  */
7554 cpupart_t *
7555 mac_pset_find(mac_resource_props_t *mrp, boolean_t *use_default)
7556 {
7557         pool_t          *pool;
7558         cpupart_t       *cpupart;

7560         *use_default = B_FALSE;

7562         /* CPUs property is set */
7563         if (mrp->mrp_mask & MRP_CPUS)
7564                 return (NULL);

7566         ASSERT(pool_lock_held());

7568         /* Pools are disabled, no pset */
7569         if (pool_state == POOL_DISABLED)
7570                 return (NULL);

7572         /* Pools property is set */
7573         if (mrp->mrp_mask & MRP_POOL) {
7574                 if ((pool = pool_lookup_pool_by_name(mrp->mrp_pool)) == NULL) {
7575                         /* Pool not found */
7576                         DTRACE_PROBE1(mac_pset_find_no_pool, char *,
7577                             mrp->mrp_pool);
7578                         *use_default = B_TRUE;
7579                         pool = pool_default;
7580                 }
7581         /* Pools property is not set */
7582         } else {
7583                 *use_default = B_TRUE;
7584                 pool = pool_default;
7585         }

7587         /* Find the CPU pset that corresponds to the pool */
7588         mutex_enter(&cpu_lock);
7589         if ((cpupart = cpupart_find(pool->pool_pset->pset_id)) == NULL) {
7590                 DTRACE_PROBE1(mac_find_pset_no_pset, psetid_t,
```

```
7591                    pool->pool_pset->pset_id);
7592        }
7593        mutex_exit(&cpu_lock);

7595        return (cpupart);
7596 }

7598 void
7599 mac_set_pool_effective(boolean_t use_default, cpupart_t *cpupart,
7600     mac_resource_props_t *mrp, mac_resource_props_t *emrp)
7601 {
7602        ASSERT(pool_lock_held());

7604        if (cpupart != NULL) {
7605               emrp->mrp_mask |= MRP_POOL;
7606               if (use_default) {
7607                      (void) strcpy(emrp->mrp_pool,
7608                          "pool_default");
7609               } else {
7610                      ASSERT(strlen(mrp->mrp_pool) != 0);
7611                      (void) strcpy(emrp->mrp_pool,
7612                          mrp->mrp_pool);
7613               }
7614        } else {
7615               emrp->mrp_mask &= ~MRP_POOL;
7616               bzero(emrp->mrp_pool, MAXPATHLEN);
7617        }
7618 }

7620 struct mac_pool_arg {
7621        char            mpa_poolname[MAXPATHLEN];
7622        pool_event_t    mpa_what;
7623 };

7625 /*ARGSUSED*/
7626 static uint_t
7627 mac_pool_link_update(mod_hash_key_t key, mod_hash_val_t *val, void *arg)
7628 {
7629        struct mac_pool_arg     *mpa = arg;
7630        mac_impl_t              *mip = (mac_impl_t *)val;
7631        mac_client_impl_t       *mcip;
7632        mac_resource_props_t    *mrp, *emrp;
7633        boolean_t               pool_update = B_FALSE;
7634        boolean_t               pool_clear = B_FALSE;
7635        boolean_t               use_default = B_FALSE;
7636        cpupart_t               *cpupart = NULL;

7638        mrp = kmem_zalloc(sizeof (*mrp), KM_SLEEP);
7639        i_mac_perim_enter(mip);
7640        for (mcip = mip->mi_clients_list; mcip != NULL;
7641            mcip = mcip->mci_client_next) {
7642               pool_update = B_FALSE;
7643               pool_clear = B_FALSE;
7644               use_default = B_FALSE;
7645               mac_client_get_resources((mac_client_handle_t)mcip, mrp);
7646               emrp = MCIP_EFFECTIVE_PROPS(mcip);

7648               /*
7649                * When pools are enabled
7650                */
7651               if ((mpa->mpa_what == POOL_E_ENABLE) &&
7652                   ((mrp->mrp_mask & MRP_CPUS) == 0)) {
7653                      mrp->mrp_mask |= MRP_POOL;
7654                      pool_update = B_TRUE;
7655               }
```



```
7657               /*
7658                * When pools are disabled
7659                */
7660               if ((mpa->mpa_what == POOL_E_DISABLE) &&
7661                   ((mrp->mrp_mask & MRP_CPUS) == 0)) {
7662                      mrp->mrp_mask |= MRP_POOL;
7663                      pool_clear = B_TRUE;
7664               }

7666               /*
7667                * Look for links with the pool property set and the poolname
7668                * matching the one which is changing.
7669                */
7670               if (strcmp(mrp->mrp_pool, mpa->mpa_poolname) == 0) {
7671                      /*
7672                       * The pool associated with the link has changed.
7673                       */
7674                      if (mpa->mpa_what == POOL_E_CHANGE) {
7675                             mrp->mrp_mask |= MRP_POOL;
7676                             pool_update = B_TRUE;
7677                      }
7678               }

7680               /*
7681                * This link is associated with pool_default and
7682                * pool_default has changed.
7683                */
7684               if ((mpa->mpa_what == POOL_E_CHANGE) &&
7685                   (strcmp(emrp->mrp_pool, "pool_default") == 0) &&
7686                   (strcmp(mpa->mpa_poolname, "pool_default") == 0)) {
7687                      mrp->mrp_mask |= MRP_POOL;
7688                      pool_update = B_TRUE;
7689               }

7691               /*
7692                * Get new list of cpus for the pool, bind network
7693                * threads to new list of cpus and update resources.
7694                */
7695               if (pool_update) {
7696                      if (MCIP_DATAPATH_SETUP(mcip)) {
7697                             pool_lock();
7698                             cpupart = mac_pset_find(mrp, &use_default);
7699                             mac_fanout_setup(mcip, mcip->mci_flent, mrp,
7700                                 mac_rx_deliver, mcip, NULL, cpupart);
7701                             mac_set_pool_effective(use_default, cpupart,
7702                                 mrp, emrp);
7703                             pool_unlock();
7704                      }
7705                      mac_update_resources(mrp, MCIP_RESOURCE_PROPS(mcip),
7706                          B_FALSE);
7707               }

7709               /*
7710                * Clear the effective pool and bind network threads
7711                * to any available CPU.
7712                */
7713               if (pool_clear) {
7714                      if (MCIP_DATAPATH_SETUP(mcip)) {
7715                             emrp->mrp_mask &= ~MRP_POOL;
7716                             bzero(emrp->mrp_pool, MAXPATHLEN);
7717                             mac_fanout_setup(mcip, mcip->mci_flent, mrp,
7718                                 mac_rx_deliver, mcip, NULL, NULL);
7719                      }
7720                      mac_update_resources(mrp, MCIP_RESOURCE_PROPS(mcip),
7721                          B_FALSE);
7722               }
```

```
7723         }
7724         i_mac_perim_exit(mip);
7725         kmem_free(mrp, sizeof (*mrp));
7726         return (MH_WALK_CONTINUE);
7727 }

7729 static void
7730 mac_pool_update(void *arg)
7731 {
7732         mod_hash_walk(i_mac_impl_hash, mac_pool_link_update, arg);
7733         kmem_free(arg, sizeof (struct mac_pool_arg));
7734 }

7736 /*
7737  * Callback function to be executed when a noteworthy pool event
7738  * takes place.
7739  */
7740 /* ARGSUSED */
7741 static void
7742 mac_pool_event_cb(pool_event_t what, poolid_t id, void *arg)
7743 {
7744         pool_t                  *pool;
7745         char                    *poolname = NULL;
7746         struct mac_pool_arg     *mpa;

7748         pool_lock();
7749         mpa = kmem_zalloc(sizeof (struct mac_pool_arg), KM_SLEEP);

7751         switch (what) {
7752         case POOL_E_ENABLE:
7753         case POOL_E_DISABLE:
7754                 break;

7756         case POOL_E_CHANGE:
7757                 pool = pool_lookup_pool_by_id(id);
7758                 if (pool == NULL) {
7759                         kmem_free(mpa, sizeof (struct mac_pool_arg));
7760                         pool_unlock();
7761                         return;
7762                 }
7763                 pool_get_name(pool, &poolname);
7764                 (void) strlcpy(mpa->mpa_poolname, poolname,
7765                     sizeof (mpa->mpa_poolname));
7766                 break;

7768         default:
7769                 kmem_free(mpa, sizeof (struct mac_pool_arg));
7770                 pool_unlock();
7771                 return;
7772         }
7773         pool_unlock();

7775         mpa->mpa_what = what;

7777         mac_pool_update(mpa);
7778 }

7780 /*
7781  * Set effective rings property. This could be called from datapath_setup/
7782  * datapath_teardown or set-linkprop.
7783  * If the group is reserved we just go ahead and set the effective rings.
7784  * Additionally, for TX this could mean the default  group has lost/gained
7785  * some rings, so if the default group is reserved, we need to adjust the
7786  * effective rings for the default group clients. For RX, if we are working
7787  * with the non-default group, we just need * to reset the effective props
7788  * for the default group clients.
```



```
7789  */
7790 void
7791 mac_set_rings_effective(mac_client_impl_t *mcip)
7792 {
7793         mac_impl_t              *mip = mcip->mci_mip;
7794         mac_group_t             *grp;
7795         mac_group_t             *defgrp;
7796         flow_entry_t            *flent = mcip->mci_flent;
7797         mac_resource_props_t    *emrp = MCIP_EFFECTIVE_PROPS(mcip);
7798         mac_grp_client_t        *mgcp;
7799         mac_client_impl_t       *gmcip;

7801         grp = flent->fe_rx_ring_group;
7802         if (grp != NULL) {
7803                 defgrp = MAC_DEFAULT_RX_GROUP(mip);
7804                 /*
7805                  * If we have reserved a group, set the effective rings
7806                  * to the ring count in the group.
7807                  */
7808                 if (grp->mrg_state == MAC_GROUP_STATE_RESERVED) {
7809                         emrp->mrp_mask |= MRP_RX_RINGS;
7810                         emrp->mrp_nrxrings = grp->mrg_cur_count;
7811                 }

7813                 /*
7814                  * We go through the clients in the shared group and
7815                  * reset the effective properties. It is possible this
7816                  * might have already been done for some client (i.e.
7817                  * if some client is being moved to a group that is
7818                  * already shared). The case where the default group is
7819                  * RESERVED is taken care of above (note in the RX side if
7820                  * there is a non-default group, the default group is always
7821                  * SHARED).
7822                  */
7823                 if (grp != defgrp || grp->mrg_state == MAC_GROUP_STATE_SHARED) {
7824                         if (grp->mrg_state == MAC_GROUP_STATE_SHARED)
7825                                 mgcp = grp->mrg_clients;
7826                         else
7827                                 mgcp = defgrp->mrg_clients;
7828                         while (mgcp != NULL) {
7829                                 gmcip = mgcp->mgc_client;
7830                                 emrp = MCIP_EFFECTIVE_PROPS(gmcip);
7831                                 if (emrp->mrp_mask & MRP_RX_RINGS) {
7832                                         emrp->mrp_mask &= ~MRP_RX_RINGS;
7833                                         emrp->mrp_nrxrings = 0;
7834                                 }
7835                                 mgcp = mgcp->mgc_next;
7836                         }
7837                 }
7838         }

7840         /* Now the TX side */
7841         grp = flent->fe_tx_ring_group;
7842         if (grp != NULL) {
7843                 defgrp = MAC_DEFAULT_TX_GROUP(mip);

7845                 if (grp->mrg_state == MAC_GROUP_STATE_RESERVED) {
7846                         emrp->mrp_mask |= MRP_TX_RINGS;
7847                         emrp->mrp_ntxrings = grp->mrg_cur_count;
7848                 } else if (grp->mrg_state == MAC_GROUP_STATE_SHARED) {
7849                         mgcp = grp->mrg_clients;
7850                         while (mgcp != NULL) {
7851                                 gmcip = mgcp->mgc_client;
7852                                 emrp = MCIP_EFFECTIVE_PROPS(gmcip);
7853                                 if (emrp->mrp_mask & MRP_TX_RINGS) {
7854                                         emrp->mrp_mask &= ~MRP_TX_RINGS;
```

```
7855                                        emrp->mrp_ntxrings = 0;
7856                                }
7857                                mgcp = mgcp->mgc_next;
7858                        }
7859                }

7861                /*
7862                 * If the group is not the default group and the default
7863                 * group is reserved, the ring count in the default group
7864                 * might have changed, update it.
7865                 */
7866                if (grp != defgrp &&
7867                    defgrp->mrg_state == MAC_GROUP_STATE_RESERVED) {
7868                        gmcip = MAC_GROUP_ONLY_CLIENT(defgrp);
7869                        emrp = MCIP_EFFECTIVE_PROPS(gmcip);
7870                        emrp->mrp_ntxrings = defgrp->mrg_cur_count;
7871                }
7872        }
7873        emrp = MCIP_EFFECTIVE_PROPS(mcip);
7874 }

7876 /*
7877  * Check if the primary is in the default group. If so, see if we
7878  * can give it a an exclusive group now that another client is
7879  * being configured. We take the primary out of the default group
7880  * because the multicast/broadcast packets for the all the clients
7881  * will land in the default ring in the default group which means
7882  * any client in the default group, even if it is the only on in
7883  * the group, will lose exclusive access to the rings, hence
7884  * polling.
7885  */
7886 mac_client_impl_t *
7887 mac_check_primary_relocation(mac_client_impl_t *mcip, boolean_t rxhw)
7888 {
7889        mac_impl_t              *mip = mcip->mci_mip;
7890        mac_group_t             *defgrp = MAC_DEFAULT_RX_GROUP(mip);
7891        flow_entry_t            *flent = mcip->mci_flent;
7892        mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);
7893        uint8_t                 *mac_addr;
7894        mac_group_t             *ngrp;

7896        /*
7897         * Check if the primary is in the default group, if not
7898         * or if it is explicitly configured to be in the default
7899         * group OR set the RX rings property, return.
7900         */
7901        if (flent->fe_rx_ring_group != defgrp || mrp->mrp_mask & MRP_RX_RINGS)
7902                return (NULL);

7904        /*
7905         * If the new client needs an exclusive group and we
7906         * don't have another for the primary, return.
7907         */
7908        if (rxhw && mip->mi_rxhwclnt_avail < 2)
7909                return (NULL);

7911        mac_addr = flent->fe_flow_desc.fd_dst_mac;
7912        /*
7913         * We call this when we are setting up the datapath for
7914         * the first non-primary.
7915         */
7916        ASSERT(mip->mi_nactiveclients == 2);
7917        /*
7918         * OK, now we have the primary that needs to be relocated.
7919         */
7920        ngrp =  mac_reserve_rx_group(mcip, mac_addr, B_TRUE);
```



```
7921        if (ngrp == NULL)
7922                return (NULL);
7923        if (mac_rx_switch_group(mcip, defgrp, ngrp) != 0) {
7924                mac_stop_group(ngrp);
7925                return (NULL);
7926        }
7927        return (mcip);
7928 }
```

```
**********************************************************
  148147 Thu Feb 20 18:59:06 2014
new/usr/src/uts/common/io/mac/mac_client.c
2553 mac address should be a dladm link property
**********************************************************
______unchanged_portion_omitted_

2344 /*
2345  * Add a new unicast address to the MAC client.
2346  *
2347  * The MAC address can be specified either by value, or the MAC client
2348  * can specify that it wants to use the primary MAC address of the
2349  * underlying MAC. See the introductory comments at the beginning
2350  * of this file for more information on primary MAC addresses.
2350  * of this file for more more information on primary MAC addresses.
2351  *
2352  * Note also the tuple (MAC address, VID) must be unique
2353  * for the MAC clients defined on top of the same underlying MAC
2354  * instance, unless the MAC_UNICAST_NODUPCHECK is specified.
2355  *
2356  * In no case can a client use the PVID for the MAC, if the MAC has one set.
2357  */
2358 int
2359 i_mac_unicast_add(mac_client_handle_t mch, uint8_t *mac_addr, uint16_t flags,
2360     mac_unicast_handle_t *mah, uint16_t vid, mac_diag_t *diag)
2361 {
2362         mac_client_impl_t       *mcip = (mac_client_impl_t *)mch;
2363         mac_impl_t              *mip = mcip->mci_mip;
2364         int                     err;
2365         uint_t                  mac_len = mip->mi_type->mt_addr_length;
2366         boolean_t               check_dups = !(flags & MAC_UNICAST_NODUPCHECK);
2367         boolean_t               fastpath_disabled = B_FALSE;
2368         boolean_t               is_primary = (flags & MAC_UNICAST_PRIMARY);
2369         boolean_t               is_unicast_hw = (flags & MAC_UNICAST_HW);
2370         mac_resource_props_t    *mrp;
2371         boolean_t               passive_client = B_FALSE;
2372         mac_unicast_impl_t      *muip;
2373         boolean_t               is_vnic_primary =
2374             (flags & MAC_UNICAST_VNIC_PRIMARY);

2376         /* when VID is non-zero, the underlying MAC can not be VNIC */
2377         ASSERT(!((mip->mi_state_flags & MIS_IS_VNIC) && (vid != 0)));

2379         /*
2380          * Can't unicast add if the client asked only for minimal datapath
2381          * setup.
2382          */
2383         if (mcip->mci_state_flags & MCIS_NO_UNICAST_ADDR)
2384                 return (ENOTSUP);

2386         /*
2387          * Check for an attempted use of the current Port VLAN ID, if enabled.
2388          * No client may use it.
2389          */
2390         if (mip->mi_pvid != 0 && vid == mip->mi_pvid)
2391                 return (EBUSY);

2393         /*
2394          * Check whether it's the primary client and flag it.
2395          */
2396         if (!(mcip->mci_state_flags & MCIS_IS_VNIC) && is_primary && vid == 0)
2397                 mcip->mci_flags |= MAC_CLIENT_FLAGS_PRIMARY;

2399         /*
2400          * is_vnic_primary is true when we come here as a VLAN VNIC
2401          * which uses the primary mac client's address but with a non-zero
```



```
2402          * VID. In this case the MAC address is not specified by an upper
2403          * MAC client.
2404          */
2405         if ((mcip->mci_state_flags & MCIS_IS_VNIC) && is_primary &&
2406             !is_vnic_primary) {
2407                 /*
2408                  * The address is being set by the upper MAC client
2409                  * of a VNIC. The MAC address was already set by the
2410                  * VNIC driver during VNIC creation.
2411                  *
2412                  * Note: a VNIC has only one MAC address. We return
2413                  * the MAC unicast address handle of the lower MAC client
2414                  * corresponding to the VNIC. We allocate a new entry
2415                  * which is flagged appropriately, so that mac_unicast_remove()
2416                  * doesn't attempt to free the original entry that
2417                  * was allocated by the VNIC driver.
2418                  */
2419                 ASSERT(mcip->mci_unicast != NULL);

2421                 /* Check for VLAN flags, if present */
2422                 if ((flags & MAC_UNICAST_TAG_DISABLE) != 0)
2423                         mcip->mci_state_flags |= MCIS_TAG_DISABLE;

2425                 if ((flags & MAC_UNICAST_STRIP_DISABLE) != 0)
2426                         mcip->mci_state_flags |= MCIS_STRIP_DISABLE;

2428                 if ((flags & MAC_UNICAST_DISABLE_TX_VID_CHECK) != 0)
2429                         mcip->mci_state_flags |= MCIS_DISABLE_TX_VID_CHECK;

2431                 /*
2432                  * Ensure that the primary unicast address of the VNIC
2433                  * is added only once unless we have the
2434                  * MAC_CLIENT_FLAGS_MULTI_PRIMARY set (and this is not
2435                  * a passive MAC client).
2436                  */
2437                 if ((mcip->mci_flags & MAC_CLIENT_FLAGS_VNIC_PRIMARY) != 0) {
2438                         if ((mcip->mci_flags &
2439                             MAC_CLIENT_FLAGS_MULTI_PRIMARY) == 0 ||
2440                             (mcip->mci_flags &
2441                             MAC_CLIENT_FLAGS_PASSIVE_PRIMARY) != 0) {
2442                                 return (EBUSY);
2443                         }
2444                         mcip->mci_flags |= MAC_CLIENT_FLAGS_PASSIVE_PRIMARY;
2445                         passive_client = B_TRUE;
2446                 }

2448                 mcip->mci_flags |= MAC_CLIENT_FLAGS_VNIC_PRIMARY;

2450                 /*
2451                  * Create a handle for vid 0.
2452                  */
2453                 ASSERT(vid == 0);
2454                 muip = kmem_zalloc(sizeof (mac_unicast_impl_t), KM_SLEEP);
2455                 muip->mui_vid = vid;
2456                 *mah = (mac_unicast_handle_t)muip;
2457                 /*
2458                  * This will be used by the caller to defer setting the
2459                  * rx functions.
2460                  */
2461                 if (passive_client)
2462                         return (EAGAIN);
2463                 return (0);
2464         }

2466         /* primary MAC clients cannot be opened on top of anchor VNICs */
2467         if ((is_vnic_primary || is_primary) &&
```

```
2468             i_mac_capab_get((mac_handle_t)mip, MAC_CAPAB_ANCHOR_VNIC, NULL)) {
2469                 return (ENXIO);
2470         }

2472         /*
2473          * If this is a VNIC/VLAN, disable softmac fast-path.
2474          */
2475         if (mcip->mci_state_flags & MCIS_IS_VNIC) {
2476                 err = mac_fastpath_disable((mac_handle_t)mip);
2477                 if (err != 0)
2478                         return (err);
2479                 fastpath_disabled = B_TRUE;
2480         }

2482         /*
2483          * Return EBUSY if:
2484          *  - there is an exclusively active mac client exists.
2485          *  - this is an exclusive active mac client but
2486          *      a. there is already active mac clients exist, or
2487          *      b. fastpath streams are already plumbed on this legacy device
2488          *  - the mac creator has disallowed active mac clients.
2489          */
2490         if (mip->mi_state_flags & (MIS_EXCLUSIVE|MIS_NO_ACTIVE)) {
2491                 if (fastpath_disabled)
2492                         mac_fastpath_enable((mac_handle_t)mip);
2493                 return (EBUSY);
2494         }

2496         if (mcip->mci_state_flags & MCIS_EXCLUSIVE) {
2497                 ASSERT(!fastpath_disabled);
2498                 if (mip->mi_nactiveclients != 0)
2499                         return (EBUSY);

2501                 if ((mip->mi_state_flags & MIS_LEGACY) &&
2502                     !(mip->mi_capab_legacy.ml_active_set(mip->mi_driver))) {
2503                         return (EBUSY);
2504                 }
2505                 mip->mi_state_flags |= MIS_EXCLUSIVE;
2506         }

2508         mrp = kmem_zalloc(sizeof (*mrp), KM_SLEEP);
2509         if (is_primary && !(mcip->mci_state_flags & (MCIS_IS_VNIC |
2510             MCIS_IS_AGGR_PORT))) {
2511                 /*
2512                  * Apply the property cached in the mac_impl_t to the primary
2513                  * mac client. If the mac client is a VNIC or an aggregation
2514                  * port, its property should be set in the mcip when the
2515                  * VNIC/aggr was created.
2516                  */
2517                 mac_get_resources((mac_handle_t)mip, mrp);
2518                 (void) mac_client_set_resources(mch, mrp);
2519         } else if (mcip->mci_state_flags & MCIS_IS_VNIC) {
2520                 /*
2521                  * This is a primary VLAN client, we don't support
2522                  * specifying rings property for this as it inherits the
2523                  * rings property from its MAC.
2524                  */
2525                 if (is_vnic_primary) {
2526                         mac_resource_props_t    *vmrp;

2528                         vmrp = MCIP_RESOURCE_PROPS(mcip);
2529                         if (vmrp->mrp_mask & MRP_RX_RINGS ||
2530                             vmrp->mrp_mask & MRP_TX_RINGS) {
2531                                 if (fastpath_disabled)
2532                                         mac_fastpath_enable((mac_handle_t)mip);
2533                                 kmem_free(mrp, sizeof (*mrp));
```



```
2534                                 return (ENOTSUP);
2535                         }
2536                         /*
2537                          * Additionally we also need to inherit any
2538                          * rings property from the MAC.
2539                          */
2540                         mac_get_resources((mac_handle_t)mip, mrp);
2541                         if (mrp->mrp_mask & MRP_RX_RINGS) {
2542                                 vmrp->mrp_mask |= MRP_RX_RINGS;
2543                                 vmrp->mrp_nrxrings = mrp->mrp_nrxrings;
2544                         }
2545                         if (mrp->mrp_mask & MRP_TX_RINGS) {
2546                                 vmrp->mrp_mask |= MRP_TX_RINGS;
2547                                 vmrp->mrp_ntxrings = mrp->mrp_ntxrings;
2548                         }
2549                 }
2550                 bcopy(MCIP_RESOURCE_PROPS(mcip), mrp, sizeof (*mrp));
2551         }

2553         muip = kmem_zalloc(sizeof (mac_unicast_impl_t), KM_SLEEP);
2554         muip->mui_vid = vid;

2556         if (is_primary || is_vnic_primary) {
2557                 mac_addr = mip->mi_addr;
2558         } else {

2560                 /*
2561                  * Verify the validity of the specified MAC addresses value.
2562                  */
2563                 if (!mac_unicst_verify((mac_handle_t)mip, mac_addr, mac_len)) {
2564                         *diag = MAC_DIAG_MACADDR_INVALID;
2565                         err = EINVAL;
2566                         goto bail_out;
2567                 }

2569                 /*
2570                  * Make sure that the specified MAC address is different
2571                  * than the unicast MAC address of the underlying NIC.
2572                  */
2573                 if (check_dups && bcmp(mip->mi_addr, mac_addr, mac_len) == 0) {
2574                         *diag = MAC_DIAG_MACADDR_NIC;
2575                         err = EINVAL;
2576                         goto bail_out;
2577                 }
2578         }

2580         /*
2581          * Set the flags here so that if this is a passive client, we
2582          * can return  and set it when we call mac_client_datapath_setup
2583          * when this becomes the active client. If we defer to using these
2584          * flags to mac_client_datapath_setup, then for a passive client,
2585          * we'd have to store the flags somewhere (probably fe_flags)
2586          * and then use it.
2587          */
2588         if (!MCIP_DATAPATH_SETUP(mcip)) {
2589                 if (is_unicast_hw) {
2590                         /*
2591                          * The client requires a hardware MAC address slot
2592                          * for that unicast address. Since we support only
2593                          * one unicast MAC address per client, flag the
2594                          * MAC client itself.
2595                          */
2596                         mcip->mci_state_flags |= MCIS_UNICAST_HW;
2597                 }

2599                 /* Check for VLAN flags, if present */
```

```
2600                 if ((flags & MAC_UNICAST_TAG_DISABLE) != 0)
2601                         mcip->mci_state_flags |= MCIS_TAG_DISABLE;

2603                 if ((flags & MAC_UNICAST_STRIP_DISABLE) != 0)
2604                         mcip->mci_state_flags |= MCIS_STRIP_DISABLE;

2606                 if ((flags & MAC_UNICAST_DISABLE_TX_VID_CHECK) != 0)
2607                         mcip->mci_state_flags |= MCIS_DISABLE_TX_VID_CHECK;
2608         } else {
2609                 /*
2610                  * Assert that the specified flags are consistent with the
2611                  * flags specified by previous calls to mac_unicast_add().
2612                  */
2613                 ASSERT(((flags & MAC_UNICAST_TAG_DISABLE) != 0 &&
2614                     (mcip->mci_state_flags & MCIS_TAG_DISABLE) != 0) ||
2615                     ((flags & MAC_UNICAST_TAG_DISABLE) == 0 &&
2616                     (mcip->mci_state_flags & MCIS_TAG_DISABLE) == 0));

2618                 ASSERT(((flags & MAC_UNICAST_STRIP_DISABLE) != 0 &&
2619                     (mcip->mci_state_flags & MCIS_STRIP_DISABLE) != 0) ||
2620                     ((flags & MAC_UNICAST_STRIP_DISABLE) == 0 &&
2621                     (mcip->mci_state_flags & MCIS_STRIP_DISABLE) == 0));

2623                 ASSERT(((flags & MAC_UNICAST_DISABLE_TX_VID_CHECK) != 0 &&
2624                     (mcip->mci_state_flags & MCIS_DISABLE_TX_VID_CHECK) != 0) ||
2625                     ((flags & MAC_UNICAST_DISABLE_TX_VID_CHECK) == 0 &&
2626                     (mcip->mci_state_flags & MCIS_DISABLE_TX_VID_CHECK) == 0));

2628                 /*
2629                  * Make sure the client is consistent about its requests
2630                  * for MAC addresses. I.e. all requests from the clients
2631                  * must have the MAC_UNICAST_HW flag set or clear.
2632                  */
2633                 if ((mcip->mci_state_flags & MCIS_UNICAST_HW) != 0 &&
2634                     !is_unicast_hw ||
2635                     (mcip->mci_state_flags & MCIS_UNICAST_HW) == 0 &&
2636                     is_unicast_hw) {
2637                         err = EINVAL;
2638                         goto bail_out;
2639                 }
2640         }
2641         /*
2642          * Make sure the MAC address is not already used by
2643          * another MAC client defined on top of the same
2644          * underlying NIC. Unless we have MAC_CLIENT_FLAGS_MULTI_PRIMARY
2645          * set when we allow a passive client to be present which will
2646          * be activated when the currently active client goes away - this
2647          * works only with primary addresses.
2648          */
2649         if ((check_dups || is_primary || is_vnic_primary) &&
2650             mac_addr_in_use(mip, mac_addr, vid)) {
2651                 /*
2652                  * Must have set the multiple primary address flag when
2653                  * we did a mac_client_open AND this should be a primary
2654                  * MAC client AND there should not already be a passive
2655                  * primary. If all is true then we let this succeed
2656                  * even if the address is a dup.
2657                  */
2658                 if ((mcip->mci_flags & MAC_CLIENT_FLAGS_MULTI_PRIMARY) == 0 ||
2659                     (mcip->mci_flags & MAC_CLIENT_FLAGS_PRIMARY) == 0 ||
2660                     mac_get_passive_primary_client(mip) != NULL) {
2661                         *diag = MAC_DIAG_MACADDR_INUSE;
2662                         err = EEXIST;
2663                         goto bail_out;
2664                 }
2665                 ASSERT((mcip->mci_flags &
```



```
2666                     MAC_CLIENT_FLAGS_PASSIVE_PRIMARY) == 0);
2667                 mcip->mci_flags |= MAC_CLIENT_FLAGS_PASSIVE_PRIMARY;
2668                 kmem_free(mrp, sizeof (*mrp));

2670                 /*
2671                  * Stash the unicast address handle, we will use it when
2672                  * we set up the passive client.
2673                  */
2674                 mcip->mci_p_unicast_list = muip;
2675                 *mah = (mac_unicast_handle_t)muip;
2676                 return (0);
2677         }

2679         err = mac_client_datapath_setup(mcip, vid, mac_addr, mrp,
2680             is_primary || is_vnic_primary, muip);
2681         if (err != 0)
2682                 goto bail_out;

2684         kmem_free(mrp, sizeof (*mrp));
2685         *mah = (mac_unicast_handle_t)muip;
2686         return (0);

2688 bail_out:
2689         if (fastpath_disabled)
2690                 mac_fastpath_enable((mac_handle_t)mip);
2691         if (mcip->mci_state_flags & MCIS_EXCLUSIVE) {
2692                 mip->mi_state_flags &= ~MIS_EXCLUSIVE;
2693                 if (mip->mi_state_flags & MIS_LEGACY) {
2694                         mip->mi_capab_legacy.ml_active_clear(
2695                             mip->mi_driver);
2696                 }
2697         }
2698         kmem_free(mrp, sizeof (*mrp));
2699         kmem_free(muip, sizeof (mac_unicast_impl_t));
2700         return (err);
2701 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   19026 Thu Feb 20 18:59:07 2014
new/usr/src/uts/common/sys/mac.h
2553 mac address should be a dladm link property
**********************************************************
______unchanged_portion_omitted_

typedef struct mac_addrprop_s {
        uint32_t        ma_len;
        uint8_t         ma_addr[MAXMACADDRLEN];
} mac_addrprop_t;

#endif /* ! codereview */
#define MAXLINKPROPNAME         256             /* max property name len */

/*
 * Public properties.
 *
 * Note that there are 2 sets of parameters: the *_EN_* values are
 * those that the Administrator configures for autonegotiation. The
 * _ADV_* values are those that are currently exposed over the wire.
 */
typedef enum {
        MAC_PROP_DUPLEX = 0x00000001,
        MAC_PROP_SPEED,
        MAC_PROP_STATUS,
        MAC_PROP_AUTONEG,
        MAC_PROP_EN_AUTONEG,
        MAC_PROP_MTU,
        MAC_PROP_ZONE,
        MAC_PROP_AUTOPUSH,
        MAC_PROP_FLOWCTRL,
        MAC_PROP_ADV_1000FDX_CAP,
        MAC_PROP_EN_1000FDX_CAP,
        MAC_PROP_ADV_1000HDX_CAP,
        MAC_PROP_EN_1000HDX_CAP,
        MAC_PROP_ADV_100FDX_CAP,
        MAC_PROP_EN_100FDX_CAP,
        MAC_PROP_ADV_100HDX_CAP,
        MAC_PROP_EN_100HDX_CAP,
        MAC_PROP_ADV_10FDX_CAP,
        MAC_PROP_EN_10FDX_CAP,
        MAC_PROP_ADV_10HDX_CAP,
        MAC_PROP_EN_10HDX_CAP,
        MAC_PROP_ADV_100T4_CAP,
        MAC_PROP_EN_100T4_CAP,
        MAC_PROP_IPTUN_HOPLIMIT,
        MAC_PROP_IPTUN_ENCAPLIMIT,
        MAC_PROP_WL_ESSID,
        MAC_PROP_WL_BSSID,
        MAC_PROP_WL_BSSTYPE,
        MAC_PROP_WL_LINKSTATUS,
        MAC_PROP_WL_DESIRED_RATES,
        MAC_PROP_WL_SUPPORTED_RATES,
        MAC_PROP_WL_AUTH_MODE,
        MAC_PROP_WL_ENCRYPTION,
        MAC_PROP_WL_RSSI,
        MAC_PROP_WL_PHY_CONFIG,
        MAC_PROP_WL_CAPABILITY,
        MAC_PROP_WL_WPA,
        MAC_PROP_WL_SCANRESULTS,
        MAC_PROP_WL_POWER_MODE,
        MAC_PROP_WL_RADIO,
        MAC_PROP_WL_ESS_LIST,
        MAC_PROP_WL_KEY_TAB,
        MAC_PROP_WL_CREATE_IBSS,
```



```
        MAC_PROP_WL_SETOPTIE,
        MAC_PROP_WL_DELKEY,
        MAC_PROP_WL_KEY,
        MAC_PROP_WL_MLME,
        MAC_PROP_TAGMODE,
        MAC_PROP_ADV_10GFDX_CAP,
        MAC_PROP_EN_10GFDX_CAP,
        MAC_PROP_PVID,
        MAC_PROP_LLIMIT,
        MAC_PROP_LDECAY,
        MAC_PROP_RESOURCE,
        MAC_PROP_RESOURCE_EFF,
        MAC_PROP_RXRINGSRANGE,
        MAC_PROP_TXRINGSRANGE,
        MAC_PROP_MAX_TX_RINGS_AVAIL,
        MAC_PROP_MAX_RX_RINGS_AVAIL,
        MAC_PROP_MAX_RXHWCLNT_AVAIL,
        MAC_PROP_MAX_TXHWCLNT_AVAIL,
        MAC_PROP_IB_LINKMODE,
        MAC_PROP_MACADDRESS,
#endif /* ! codereview */
        MAC_PROP_PRIVATE = -1
} mac_prop_id_t;

/*
 * Flags to figure out r/w status of legacy ndd props.
 */
#define MAC_PROP_PERM_READ              0x0001
#define MAC_PROP_PERM_WRITE             0x0010
#define MAC_PROP_MAP_KSTAT              0x0100
#define MAC_PROP_PERM_RW                (MAC_PROP_PERM_READ|MAC_PROP_PERM_WRITE)
#define MAC_PROP_FLAGS_RK               (MAC_PROP_PERM_READ|MAC_PROP_MAP_KSTAT)

#ifdef  _KERNEL

/*
 * There are three ranges of statistics values.  0 to 1 - MAC_STAT_MIN are
 * interface statistics maintained by the mac module.  MAC_STAT_MIN to 1 -
 * MACTYPE_STAT_MIN are common MAC statistics defined by the mac module and
 * maintained by each driver.  MACTYPE_STAT_MIN and above are statistics
 * defined by MAC-Type plugins and maintained by each driver.
 */
#define MAC_STAT_MIN            1000
#define MACTYPE_STAT_MIN        2000

#define IS_MAC_STAT(stat)       \
        (stat >= MAC_STAT_MIN && stat < MACTYPE_STAT_MIN)
#define IS_MACTYPE_STAT(stat)   (stat >= MACTYPE_STAT_MIN)

/*
 * Statistics maintained by the mac module, and possibly populated as link
 * statistics.
 */
enum mac_mod_stat {
        MAC_STAT_LINK_STATE,
        MAC_STAT_LINK_UP,
        MAC_STAT_PROMISC,
        MAC_STAT_LOWLINK_STATE,
        MAC_STAT_HDROPS
};

/*
 * Do not reorder, and add only to the end of this list.
 */
enum mac_driver_stat {
        /* MIB-II stats (RFC 1213 and RFC 1573) */
```

```
261         MAC_STAT_IFSPEED = MAC_STAT_MIN,
262         MAC_STAT_MULTIRCV,
263         MAC_STAT_BRDCSTRCV,
264         MAC_STAT_MULTIXMT,
265         MAC_STAT_BRDCSTXMT,
266         MAC_STAT_NORCVBUF,
267         MAC_STAT_IERRORS,
268         MAC_STAT_UNKNOWNS,
269         MAC_STAT_NOXMTBUF,
270         MAC_STAT_OERRORS,
271         MAC_STAT_COLLISIONS,
272         MAC_STAT_RBYTES,
273         MAC_STAT_IPACKETS,
274         MAC_STAT_OBYTES,
275         MAC_STAT_OPACKETS,
276         MAC_STAT_UNDERFLOWS,
277         MAC_STAT_OVERFLOWS
278 };

280 #define MAC_NSTAT       (MAC_STAT_OVERFLOWS - MAC_STAT_IFSPEED + 1)

282 #define MAC_STAT_ISACOUNTER(_stat) (            \
283             (_stat) == MAC_STAT_MULTIRCV ||     \
284             (_stat) == MAC_STAT_BRDCSTRCV ||    \
285             (_stat) == MAC_STAT_MULTIXMT ||     \
286             (_stat) == MAC_STAT_BRDCSTXMT ||    \
287             (_stat) == MAC_STAT_NORCVBUF ||     \
288             (_stat) == MAC_STAT_IERRORS ||      \
289             (_stat) == MAC_STAT_UNKNOWNS ||     \
290             (_stat) == MAC_STAT_NOXMTBUF ||     \
291             (_stat) == MAC_STAT_OERRORS ||      \
292             (_stat) == MAC_STAT_COLLISIONS ||   \
293             (_stat) == MAC_STAT_RBYTES ||       \
294             (_stat) == MAC_STAT_IPACKETS ||     \
295             (_stat) == MAC_STAT_OBYTES ||       \
296             (_stat) == MAC_STAT_OPACKETS ||     \
297             (_stat) == MAC_STAT_UNDERFLOWS ||   \
298             (_stat) == MAC_STAT_OVERFLOWS)

300 /*
301  * Immutable information. (This may not be modified after registration).
302  */
303 typedef struct mac_info_s {
304         uint_t          mi_media;
305         uint_t          mi_nativemedia;
306         uint_t          mi_addr_length;
307         uint8_t         *mi_unicst_addr;
308         uint8_t         *mi_brdcst_addr;
309 } mac_info_t;

311 /*
312  * When VNICs are created on top of the NIC, there are two levels
313  * of MAC layer, a lower MAC, which is the MAC layer at the level of the
314  * physical NIC, and an upper MAC, which is the MAC layer at the level
315  * of the VNIC. Each VNIC maps to a MAC client at the lower MAC, and
316  * the SRS and classification is done at the lower MAC level. The upper
317  * MAC is therefore for the most part pass-through, and therefore
318  * special processing needs to be done at the upper MAC layer when
319  * dealing with a VNIC.
320  *
321  * This capability allows the MAC layer to detect when a VNIC is being
322  * access, and implement the required shortcuts.
323  */

325 typedef void *(*mac_client_handle_fn_t)(void *);
```



```
327 typedef struct mac_capab_vnic_s {
328         void                    *mcv_arg;
329         mac_client_handle_fn_t  mcv_mac_client_handle;
330 } mac_capab_vnic_t;

332 typedef void (*mac_rename_fn_t)(const char *, void *);
333 typedef mblk_t *(*mac_tx_ring_fn_t)(void *, mblk_t *, uintptr_t,
334     mac_ring_handle_t *);
335 typedef struct mac_capab_aggr_s {
336         mac_rename_fn_t mca_rename_fn;
337         int (*mca_unicst)(void *, const uint8_t *);
338         mac_tx_ring_fn_t mca_find_tx_ring_fn;
339         void *mca_arg;
340 } mac_capab_aggr_t;

342 /* Bridge transmit and receive function signatures */
343 typedef mblk_t *(*mac_bridge_tx_t)(mac_handle_t, mac_ring_handle_t, mblk_t *);
344 typedef void (*mac_bridge_rx_t)(mac_handle_t, mac_resource_handle_t, mblk_t *);
345 typedef void (*mac_bridge_ref_t)(mac_handle_t, boolean_t);
346 typedef link_state_t (*mac_bridge_ls_t)(mac_handle_t, link_state_t);

348 /* must change mac_notify_cb_list[] in mac_provider.c if this is changed */
349 typedef enum {
350         MAC_NOTE_LINK,
351         MAC_NOTE_UNICST,
352         MAC_NOTE_TX,
353         MAC_NOTE_DEVPROMISC,
354         MAC_NOTE_FASTPATH_FLUSH,
355         MAC_NOTE_SDU_SIZE,
356         MAC_NOTE_DEST,
357         MAC_NOTE_MARGIN,
358         MAC_NOTE_CAPAB_CHG,
359         MAC_NOTE_LOWLINK,
360         MAC_NOTE_ALLOWED_IPS,
361         MAC_NNOTE       /* must be the last entry */
362 } mac_notify_type_t;

364 typedef void            (*mac_notify_t)(void *, mac_notify_type_t);
365 typedef void            (*mac_rx_t)(void *, mac_resource_handle_t, mblk_t *,
366                             boolean_t);
367 typedef mblk_t          *(*mac_receive_t)(void *, int);

369 /*
370  * MAC resource types
371  */
372 typedef enum {
373         MAC_RX_FIFO = 1
374 } mac_resource_type_t;

376 typedef int     (*mac_intr_enable_t)(mac_intr_handle_t);
377 typedef int     (*mac_intr_disable_t)(mac_intr_handle_t);

379 typedef struct mac_intr_s {
380         mac_intr_handle_t       mi_handle;
381         mac_intr_enable_t       mi_enable;
382         mac_intr_disable_t      mi_disable;
383         ddi_intr_handle_t       mi_ddi_handle;
384         boolean_t               mi_ddi_shared;
385 } mac_intr_t;

387 typedef struct mac_rx_fifo_s {
388         mac_resource_type_t     mrf_type;       /* MAC_RX_FIFO */
389         mac_intr_t              mrf_intr;
390         mac_receive_t           mrf_receive;
391         void                    *mrf_rx_arg;
392         uint32_t                mrf_flow_priority;
```

```
393         /*
394          * The CPU this flow is to be processed on. With intrd and future
395          * things, we should know which CPU the flow needs to be processed
396          * and get a squeue assigned on that CPU.
397          */
398         uint_t                  mrf_cpu_id;
399 } mac_rx_fifo_t;

401 #define mrf_intr_handle         mrf_intr.mi_handle
402 #define mrf_intr_enable         mrf_intr.mi_enable
403 #define mrf_intr_disable        mrf_intr.mi_disable

405 typedef union mac_resource_u {
406         mac_resource_type_t     mr_type;
407         mac_rx_fifo_t           mr_fifo;
408 } mac_resource_t;

410 typedef enum {
411         MAC_ADDRTYPE_UNICAST,
412         MAC_ADDRTYPE_MULTICAST,
413         MAC_ADDRTYPE_BROADCAST
414 } mac_addrtype_t;

416 typedef struct mac_header_info_s {
417         size_t          mhi_hdrsize;
418         size_t          mhi_pktsize;
419         const uint8_t   *mhi_daddr;
420         const uint8_t   *mhi_saddr;
421         uint32_t        mhi_origsap;
422         uint32_t        mhi_bindsap;
423         mac_addrtype_t  mhi_dsttype;
424         uint16_t        mhi_tci;
425         boolean_t       mhi_istagged;
426         boolean_t       mhi_ispvid;
427 } mac_header_info_t;

429 /*
430  * Function pointer to match dls client signature. Should be same as
431  * dls_rx_t to allow a soft ring to bypass DLS layer and call a DLS
432  * client directly.
433  */
434 typedef void            (*mac_direct_rx_t)(void *, mac_resource_handle_t,
435                                 mblk_t *, mac_header_info_t *);

437 typedef mac_resource_handle_t   (*mac_resource_add_t)(void *, mac_resource_t *);
438 typedef int                     (*mac_resource_bind_t)(void *,
439     mac_resource_handle_t, processorid_t);
440 typedef void                    (*mac_resource_remove_t)(void *, void *);
441 typedef void                    (*mac_resource_quiesce_t)(void *, void *);
442 typedef void                    (*mac_resource_restart_t)(void *, void *);
443 typedef int                     (*mac_resource_modify_t)(void *, void *,
444                                     mac_resource_t *);
445 typedef void                    (*mac_change_upcall_t)(void *, mac_direct_rx_t,
446     void *);

448 /*
449  * MAC-Type plugin interfaces
450  */

452 typedef int             (*mtops_addr_verify_t)(const void *, void *);
453 typedef boolean_t       (*mtops_sap_verify_t)(uint32_t, uint32_t *, void *);
454 typedef mblk_t          *(*mtops_header_t)(const void *, const void *,
455     uint32_t, void *, mblk_t *, size_t);
456 typedef int             (*mtops_header_info_t)(mblk_t *, void *,
457     mac_header_info_t *);
458 typedef boolean_t       (*mtops_pdata_verify_t)(void *, size_t);
```



```
459 typedef mblk_t          *(*mtops_header_modify_t)(mblk_t *, void *);
460 typedef void            (*mtops_link_details_t)(char *, size_t, mac_handle_t,
461     void *);

463 typedef struct mactype_ops_s {
464         uint_t                  mtops_ops;
465         /*
466          * mtops_unicst_verify() returns 0 if the given address is a valid
467          * unicast address, or a non-zero errno otherwise.
468          */
469         mtops_addr_verify_t     mtops_unicst_verify;
470         /*
471          * mtops_multicst_verify() returns 0 if the given address is a
472          * valid multicast address, or a non-zero errno otherwise.  If the
473          * media doesn't support multicast, ENOTSUP should be returned (for
474          * example).
475          */
476         mtops_addr_verify_t     mtops_multicst_verify;
477         /*
478          * mtops_sap_verify() returns B_TRUE if the given SAP is a valid
479          * SAP value, or B_FALSE otherwise.
480          */
481         mtops_sap_verify_t      mtops_sap_verify;
482         /*
483          * mtops_header() is used to allocate and construct a MAC header.
484          */
485         mtops_header_t          mtops_header;
486         /*
487          * mtops_header_info() is used to gather information on a given MAC
488          * header.
489          */
490         mtops_header_info_t     mtops_header_info;
491         /*
492          * mtops_pdata_verify() is used to verify the validity of MAC
493          * plugin data.  It is called by mac_register() if the driver has
494          * supplied MAC plugin data, and also by mac_pdata_update() when
495          * drivers update the data.
496          */
497         mtops_pdata_verify_t    mtops_pdata_verify;
498         /*
499          * mtops_header_cook() is an optional callback that converts (or
500          * "cooks") the given raw header (as sent by a raw DLPI consumer)
501          * into one that is appropriate to send down to the MAC driver.
502          * Following the example above, an Ethernet header sent down by a
503          * DLPI consumer would be converted to whatever header the MAC
504          * driver expects.
505          */
506         mtops_header_modify_t   mtops_header_cook;
507         /*
508          * mtops_header_uncook() is an optional callback that does the
509          * opposite of mtops_header_cook().  It "uncooks" a given MAC
510          * header (as received from the driver) for consumption by raw DLPI
511          * consumers.  For example, for a non-Ethernet plugin that wants
512          * raw DLPI consumers to be fooled into thinking that the device
513          * provides Ethernet access, this callback would modify the given
514          * mblk_t such that the MAC header is converted to an Ethernet
515          * header.
516          */
517         mtops_header_modify_t   mtops_header_uncook;
518         /*
519          * mtops_link_details() is an optional callback that provides
520          * extended information about the link state.  Its primary purpose
521          * is to provide type-specific support for syslog contents on
522          * link up events.  If no implementation is provided, then a default
523          * implementation will be used.
524          */
```

```
525         mtops_link_details_t    mtops_link_details;
526 } mactype_ops_t;

528 /*
529  * mtops_ops exists for the plugin to enumerate the optional callback
530  * entrypoints it has defined.  This allows the mac module to define
531  * additional plugin entrypoints in mactype_ops_t without breaking backward
532  * compatibility with old plugins.
533  */
534 #define MTOPS_PDATA_VERIFY      0x001
535 #define MTOPS_HEADER_COOK       0x002
536 #define MTOPS_HEADER_UNCOOK     0x004
537 #define MTOPS_LINK_DETAILS      0x008

539 /*
540  * Provide mapping for legacy ndd ioctls relevant to that mactype.
541  * Note that the ndd ioctls are obsolete, and may be removed in a future
542  * release of Solaris. The ndd ioctls are not typically used in legacy
543  * ethernet drivers. New datalink drivers of all link-types should use
544  * dladm(1m) interfaces for administering tunables and not have to provide
545  * a mapping.
546  */
547 typedef struct mac_ndd_mapping_s {
548         char            *mp_name;
549         union {
550                 mac_prop_id_t   u_id;
551                 uint_t          u_kstat;
552         } u_mp_id;
553         long            mp_minval;
554         long            mp_maxval;
555         size_t          mp_valsize;
556         int             mp_flags;
557 } mac_ndd_mapping_t;

559 #define mp_prop_id      u_mp_id.u_id
560 #define mp_kstat        u_mp_id.u_kstat

562 typedef struct mac_stat_info_s {
563         uint_t          msi_stat;
564         char            *msi_name;
565         uint_t          msi_type;       /* as defined in kstat_named_init(9F) */
566         uint64_t        msi_default;
567 } mac_stat_info_t;

569 typedef struct mactype_register_s {
570         uint_t          mtr_version;    /* set by mactype_alloc() */
571         const char      *mtr_ident;
572         mactype_ops_t   *mtr_ops;
573         uint_t          mtr_mactype;
574         uint_t          mtr_nativetype;
575         uint_t          mtr_addrlen;
576         uint8_t         *mtr_brdcst_addr;
577         mac_stat_info_t *mtr_stats;
578         size_t          mtr_statcount;
579         mac_ndd_mapping_t *mtr_mapping;
580         size_t          mtr_mappingcount;
581 } mactype_register_t;

583 /*
584  * Driver interface functions.
585  */
586 extern int                      mac_open_by_linkid(datalink_id_t,
587                                     mac_handle_t *);
588 extern int                      mac_open_by_linkname(const char *,
589                                     mac_handle_t *);
590 extern const char               *mac_name(mac_handle_t);
```



```
591 extern minor_t                  mac_minor(mac_handle_t);
592 extern minor_t                  mac_minor_hold(boolean_t);
593 extern void                     mac_minor_rele(minor_t);
594 extern void                     mac_sdu_get(mac_handle_t, uint_t *, uint_t *);
595 extern void                     mac_sdu_get2(mac_handle_t, uint_t *, uint_t *,
596                                     uint_t *);
597 extern int                      mac_maxsdu_update(mac_handle_t, uint_t);
598 extern int                      mac_maxsdu_update2(mac_handle_t, uint_t,
599                                     uint_t);
600 extern uint_t                   mac_addr_len(mac_handle_t);
601 extern int                      mac_type(mac_handle_t);
602 extern int                      mac_nativetype(mac_handle_t);

604 extern void                     mac_unicst_update(mac_handle_t,
605                                     const uint8_t *);
606 extern void                     mac_capab_update(mac_handle_t);
607 extern int                      mac_pdata_update(mac_handle_t, void *,
608                                     size_t);
609 extern boolean_t                mac_margin_update(mac_handle_t, uint32_t);
610 extern void                     mac_margin_get(mac_handle_t, uint32_t *);
611 extern int                      mac_margin_remove(mac_handle_t, uint32_t);
612 extern int                      mac_margin_add(mac_handle_t, uint32_t *,
613                                     boolean_t);
614 extern int                      mac_fastpath_disable(mac_handle_t);
615 extern void                     mac_fastpath_enable(mac_handle_t);
616 extern void                     mac_no_active(mac_handle_t);

618 extern mactype_register_t       *mactype_alloc(uint_t);
619 extern void                     mactype_free(mactype_register_t *);
620 extern int                      mactype_register(mactype_register_t *);
621 extern int                      mactype_unregister(const char *);

623 extern int                      mac_start_logusage(mac_logtype_t, uint_t);
624 extern void                     mac_stop_logusage(mac_logtype_t);

626 extern mac_handle_t             mac_get_lower_mac_handle(mac_handle_t);
627 extern boolean_t                mac_is_vnic_primary(mac_handle_t);

629 /*
630  * Packet hashing for distribution to multiple ports and rings.
631  */

633 #define MAC_PKT_HASH_L2         0x01
634 #define MAC_PKT_HASH_L3         0x02
635 #define MAC_PKT_HASH_L4         0x04

637 extern uint64_t                 mac_pkt_hash(uint_t, mblk_t *, uint8_t,
638                                     boolean_t);

640 /*
641  * Bridging linkage
642  */
643 extern void                     mac_rx_common(mac_handle_t,
644                                     mac_resource_handle_t, mblk_t *);
645 extern int                      mac_bridge_set(mac_handle_t, mac_handle_t);
646 extern void                     mac_bridge_clear(mac_handle_t, mac_handle_t);
647 extern void                     mac_bridge_vectors(mac_bridge_tx_t,
648                                     mac_bridge_rx_t, mac_bridge_ref_t,
649                                     mac_bridge_ls_t);

651 /* special case function for TRILL observability */
652 extern void                     mac_trill_snoop(mac_handle_t, mblk_t *);

654 #endif  /* _KERNEL */

656 #ifdef  __cplusplus
```

```
657 }
658 #endif

660 #endif /* _SYS_MAC_H */
```