```
**********************************************************
   20667 Wed Aug 19 07:24:52 2015
new/usr/src/uts/common/avs/ns/nsctl/nsc_gen.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/cmn_err.h>
#include <sys/errno.h>
#include <sys/ksynch.h>
#include <sys/kmem.h>
#include <sys/ddi.h>
#include <sys/varargs.h>
#if defined(DEBUG) && !defined(DS_DDICT)
#include <sys/kobj.h>
#endif

#include <sys/ncall/ncall.h>

#define __NSC_GEN__
#include "nsc_gen.h"
#include "nsc_mem.h"
#include "../nsctl.h"
#ifdef DS_DDICT
#include "../contract.h"
#endif


static kcondvar_t _nsc_delay_cv;
static kmutex_t _nsc_delay_mutex;

static nsc_service_t *_nsc_services;
static kmutex_t _nsc_svc_mutex;

static int _nsc_rmmap_inuse(nsc_rmmap_t *, ulong_t *, size_t *);

static void _nsc_sprint_dec(char **, int, int, int);
static void _nsc_sprint_hex(char **, unsigned int, int, int, int, int);

clock_t HZ;

extern nsc_rmhdr_t *_nsc_rmhdr_ptr;
```



```
void
_nsc_init_gen()
{
        HZ = drv_sectohz(1);
        HZ = drv_usectohz(1000000);
}
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   60771 Wed Aug 19 07:24:52 2015
new/usr/src/uts/common/avs/ns/sv/sv.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


1082 static int
1083 sv_prepare_unload()
1084 {
1085         int rc = 0;

1087         mutex_enter(&sv_mutex);

1089         if (sv_mod_status == SV_PREVENT_UNLOAD) {
1090                 if ((sv_ndevices != 0) || (sv_tset != NULL)) {
1091                         rc = EBUSY;
1092                 } else {
1093                         sv_mod_status = SV_ALLOW_UNLOAD;
1094                         delay(drv_sectohz(SV_WAIT_UNLOAD));
1094                         delay(SV_WAIT_UNLOAD * drv_usectohz(1000000));
1095                 }
1096         }

1098         mutex_exit(&sv_mutex);
1099         return (rc);
1100 }
______unchanged_portion_omitted_
```

```
**********************************************************
   68467 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/contract/device.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

2204 static int
2205 ct_barrier_wait_for_empty(dev_info_t *dip, int secs)
2206 {
2207         clock_t abstime;

2209         ASSERT(MUTEX_HELD(&(DEVI(dip)->devi_ct_lock)));

2211         abstime = ddi_get_lbolt() + drv_sectohz(secs);
2211         abstime = ddi_get_lbolt() + drv_usectohz(secs*1000000);
2212         while (DEVI(dip)->devi_ct_count) {
2213                 if (cv_timedwait(&(DEVI(dip)->devi_ct_cv),
2214                     &(DEVI(dip)->devi_ct_lock), abstime) == -1) {
2215                         return (-1);
2216                 }
2217         }
2218         return (0);
2219 }
______unchanged_portion_omitted_
```

```
**********************************************************
    7573 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/crypto/api/kcf_cbufcall.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 260 /*
 261  * Background processing of crypto bufcalls.
 262  */
 263 void
 264 crypto_bufcall_service(void)
 265 {
 266         callb_cpr_t     cprinfo;

 268         CALLB_CPR_INIT(&cprinfo, &cbuf_list_lock, callb_generic_cpr,
 269             "crypto_bufcall_service");

 271         mutex_enter(&cbuf_list_lock);

 273         for (;;) {
 274                 if (cbuf_list_head != NULL && KCF_GSWQ_AVAIL >= GSWQ_MINFREE) {
 275                         mutex_exit(&cbuf_list_lock);
 276                         kcf_run_cbufcalls();
 277                         mutex_enter(&cbuf_list_lock);
 278                 }

 280                 if (cbuf_list_head != NULL) {
 281                         /*
 282                          * Wait 30 seconds for queue space to become available.
 283                          * This number is reasonable as it does not cause
 284                          * much CPU overhead. We could wait on a condition
 285                          * variable and the global software dequeue routine can
 286                          * signal us. But, it adds overhead to that routine
 287                          * which we want to avoid. Also, the client is prepared
 288                          * to wait any way.
 289                          */
 290                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
 291                         mutex_exit(&cbuf_list_lock);
 292                         delay(drv_sectohz(30));
 292                         delay(30 * drv_usectohz(1000000));
 293                         mutex_enter(&cbuf_list_lock);
 294                         CALLB_CPR_SAFE_END(&cprinfo, &cbuf_list_lock);
 295                 }

 297                 /* Wait for new work to arrive */
 298                 if (cbuf_list_head == NULL) {
 299                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
 300                         cv_wait(&cbuf_list_cv, &cbuf_list_lock);
 301                         CALLB_CPR_SAFE_END(&cprinfo, &cbuf_list_lock);
 302                 }
 303         }
 304 }
______unchanged_portion_omitted_
```

```
**********************************************************
   33607 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/crypto/core/kcf_cryptoadm.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 685 /*
 686  * Called from CRYPTO_LOAD_SOFT_DISABLED ioctl.
 687  * If new_count is 0, then completely remove the entry.
 688  */
 689 int
 690 crypto_load_soft_disabled(char *name, uint_t new_count,
 691     crypto_mech_name_t *new_array)
 692 {
 693         kcf_provider_desc_t *provider = NULL;
 694         crypto_mech_name_t *prev_array;
 695         uint_t prev_count = 0;
 696         int rv;

 698         provider = kcf_prov_tab_lookup_by_name(name);
 699         if (provider != NULL) {
 700                 mutex_enter(&provider->pd_lock);
 701                 /*
 702                  * Check if any other thread is disabling or removing
 703                  * this provider. We return if this is the case.
 704                  */
 705                 if (provider->pd_state >= KCF_PROV_DISABLED) {
 706                         mutex_exit(&provider->pd_lock);
 707                         KCF_PROV_REFRELE(provider);
 708                         return (CRYPTO_BUSY);
 709                 }
 710                 provider->pd_state = KCF_PROV_DISABLED;
 711                 mutex_exit(&provider->pd_lock);

 713                 undo_register_provider(provider, B_TRUE);
 714                 KCF_PROV_REFRELE(provider);
 715                 if (provider->pd_kstat != NULL)
 716                         KCF_PROV_REFRELE(provider);

 718                 /* Wait till the existing requests complete. */
 719                 while (kcf_get_refcnt(provider, B_TRUE) > 0) {
 720                         /* wait 1 second and try again. */
 721                         delay(drv_sectohz(1));
 721                         delay(1 * drv_usectohz(1000000));
 722                 }
 723         }

 725         if (new_count == 0) {
 726                 kcf_policy_remove_by_name(name, &prev_count, &prev_array);
 727                 crypto_free_mech_list(prev_array, prev_count);
 728                 rv = CRYPTO_SUCCESS;
 729                 goto out;
 730         }

 732         /* put disabled mechanisms into policy table */
 733         if ((rv = kcf_policy_load_soft_disabled(name, new_count, new_array,
 734             &prev_count, &prev_array)) == CRYPTO_SUCCESS) {
 735                 crypto_free_mech_list(prev_array, prev_count);
 736         }

 738 out:
 739         if (provider != NULL) {
 740                 redo_register_provider(provider);
 741                 if (provider->pd_kstat != NULL)
 742                         KCF_PROV_REFHOLD(provider);
```



```
 743                 mutex_enter(&provider->pd_lock);
 744                 provider->pd_state = KCF_PROV_READY;
 745                 mutex_exit(&provider->pd_lock);
 746         } else if (rv == CRYPTO_SUCCESS) {
 747                 /*
 748                  * There are some cases where it is useful to kCF clients
 749                  * to have a provider whose mechanism is enabled now to be
 750                  * available. So, we attempt to load it here.
 751                  *
 752                  * The check, new_count < prev_count, ensures that we do this
 753                  * only in the case where a mechanism(s) is now enabled.
 754                  * This check assumes that enable and disable are separate
 755                  * administrative actions and are not done in a single action.
 756                  */
 757                 if ((new_count < prev_count) &&
 758                     (modload("crypto", name) != -1)) {
 759                         struct modctl *mcp;
 760                         boolean_t load_again = B_FALSE;

 762                         if ((mcp = mod_hold_by_name(name)) != NULL) {
 763                                 mcp->mod_loadflags |= MOD_NOAUTOUNLOAD;

 765                                 /* memory pressure may have unloaded module */
 766                                 if (!mcp->mod_installed)
 767                                         load_again = B_TRUE;
 768                                 mod_release_mod(mcp);

 770                                 if (load_again)
 771                                         (void) modload("crypto", name);
 772                         }
 773                 }
 774         }

 776         return (rv);
 777 }
______unchanged_portion_omitted_
```

```
**********************************************************
   31826 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/crypto/spi/kcf_spi.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 438 /*
 439  * This routine is used to notify the framework when a provider is being
 440  * removed.  Hardware providers call this routine in their detach routines.
 441  * Software providers call this routine in their _fini() routine.
 442  */
 443 int
 444 crypto_unregister_provider(crypto_kcf_provider_handle_t handle)
 445 {
 446         uint_t mech_idx;
 447         kcf_provider_desc_t *desc;
 448         kcf_prov_state_t saved_state;
 449         int ret = CRYPTO_SUCCESS;

 451         /* lookup provider descriptor */
 452         if ((desc = kcf_prov_tab_lookup((crypto_provider_id_t)handle)) ==
 453             NULL) {
 454                 ret = CRYPTO_UNKNOWN_PROVIDER;
 455                 goto errormsg;
 456         }

 458         mutex_enter(&desc->pd_lock);
 459         /*
 460          * Check if any other thread is disabling or removing
 461          * this provider. We return if this is the case.
 462          */
 463         if (desc->pd_state >= KCF_PROV_DISABLED) {
 464                 mutex_exit(&desc->pd_lock);
 465                 /* Release reference held by kcf_prov_tab_lookup(). */
 466                 KCF_PROV_REFRELE(desc);
 467                 ret = CRYPTO_BUSY;
 468                 goto errormsg;
 469         }

 471         saved_state = desc->pd_state;
 472         desc->pd_state = KCF_PROV_UNREGISTERING;

 474         if (saved_state == KCF_PROV_BUSY) {
 475                 /*
 476                  * The per-provider taskq threads may be waiting. We
 477                  * signal them so that they can start failing requests.
 478                  */
 479                 cv_broadcast(&desc->pd_resume_cv);
 480         }

 482         mutex_exit(&desc->pd_lock);

 484         if (desc->pd_prov_type != CRYPTO_SW_PROVIDER) {
 485                 remove_provider(desc);
 486         }

 488         if (desc->pd_prov_type != CRYPTO_LOGICAL_PROVIDER) {
 489                 /* remove the provider from the mechanisms tables */
 490                 for (mech_idx = 0; mech_idx < desc->pd_mech_list_count;
 491                     mech_idx++) {
 492                         kcf_remove_mech_provider(
 493                             desc->pd_mechanisms[mech_idx].cm_mech_name, desc);
 494                 }
 495         }
```



```
 497         /* remove provider from providers table */
 498         if (kcf_prov_tab_rem_provider((crypto_provider_id_t)handle) !=
 499             CRYPTO_SUCCESS) {
 500                 /* Release reference held by kcf_prov_tab_lookup(). */
 501                 KCF_PROV_REFRELE(desc);
 502                 ret = CRYPTO_UNKNOWN_PROVIDER;
 503                 goto errormsg;
 504         }

 506         delete_kstat(desc);

 508         if (desc->pd_prov_type == CRYPTO_SW_PROVIDER) {
 509                 /*
 510                  * Wait till the existing requests with the provider complete
 511                  * and all the holds are released. All the holds on a software
 512                  * provider are from kernel clients and the hold time
 513                  * is expected to be short. So, we won't be stuck here forever.
 514                  */
 515                 while (kcf_get_refcnt(desc, B_TRUE) > 1) {
 516                         /* wait 1 second and try again. */
 517                         delay(drv_sectohz(1));
 517                         delay(1 * drv_usectohz(1000000));
 518                 }
 519         } else {
 520                 int i;
 521                 kcf_prov_cpu_t *mp;

 523                 /*
 524                  * Wait until requests that have been sent to the provider
 525                  * complete.
 526                  */
 527                 for (i = 0; i < desc->pd_nbins; i++) {
 528                         mp = &(desc->pd_percpu_bins[i]);

 530                         mutex_enter(&mp->kp_lock);
 531                         while (mp->kp_jobcnt > 0) {
 532                                 cv_wait(&mp->kp_cv, &mp->kp_lock);
 533                         }
 534                         mutex_exit(&mp->kp_lock);
 535                 }
 536         }

 538         mutex_enter(&desc->pd_lock);
 539         desc->pd_state = KCF_PROV_UNREGISTERED;
 540         mutex_exit(&desc->pd_lock);

 542         kcf_do_notify(desc, B_FALSE);

 544         mutex_enter(&prov_tab_mutex);
 545         /* Release reference held by kcf_prov_tab_lookup(). */
 546         KCF_PROV_REFRELE(desc);

 548         if (kcf_get_refcnt(desc, B_TRUE) == 0) {
 549                 /* kcf_free_provider_desc drops prov_tab_mutex */
 550                 kcf_free_provider_desc(desc);
 551         } else {
 552                 ASSERT(desc->pd_prov_type != CRYPTO_SW_PROVIDER);
 553                 /*
 554                  * We could avoid this if /dev/crypto can proactively
 555                  * remove any holds on us from a dormant PKCS #11 app.
 556                  * For now, we check the provider table for
 557                  * KCF_PROV_UNREGISTERED entries when a provider is
 558                  * added to the table or when a provider is removed from it
 559                  * and free them when refcnt reaches zero.
 560                  */
 561                 kcf_need_provtab_walk = B_TRUE;
```

```
562                 mutex_exit(&prov_tab_mutex);
563         }

565 errormsg:
566         if (ret != CRYPTO_SUCCESS && sys_shutdown == 0) {
567                 switch (ret) {
568                 case CRYPTO_UNKNOWN_PROVIDER:
569                         cmn_err(CE_WARN, "Unknown provider \"%s\" was "
570                             "requested to unregister from the cryptographic "
571                             "framework.", desc->pd_description);
572                         break;

574                 case CRYPTO_BUSY:
575                         cmn_err(CE_WARN, "%s could not be unregistered from "
576                             "the Cryptographic Framework as it is busy.",
577                             desc->pd_description);
578                         break;

580                 default:
581                         cmn_err(CE_WARN, "%s did not unregister with the "
582                             "Cryptographic Framework. (0x%x)",
583                             desc->pd_description, ret);
584                 };
585         }

587         return (ret);
588 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11788 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/fs/dev/sdev_comm.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 105 /*
 106  * Wait for node to be created
 107  */
 108 int
 109 sdev_wait4lookup(struct sdev_node *dv, int cmd)
 110 {
 111         clock_t expire;
 112         clock_t rv;
 113         clock_t wakeup = drv_sectohz(2);
 113         clock_t wakeup = drv_usectohz(2 * 1000000);
 114         int rval = ENOENT;
 115         int is_lookup = (cmd == SDEV_LOOKUP);

 117         ASSERT(cmd == SDEV_LOOKUP || cmd == SDEV_READDIR);
 118         ASSERT(MUTEX_HELD(&dv->sdev_lookup_lock));

 120         /* tick value at which wait expires */
 121         expire = ddi_get_lbolt() +
 122             drv_sectohz(dev_node_wait_timeout);
 122             drv_usectohz(dev_node_wait_timeout * 1000000);

 124         sdcmn_err6(("wait4lookup %s %s, %ld %d\n",
 125             is_lookup ? "lookup" : "readdir",
 126             dv->sdev_name, expire - ddi_get_lbolt(), dv->sdev_state));

 128         if (SDEV_IS_LGWAITING(dv)) {
 129                 /* devfsadm nodes */
 130                 while (DEVNAME_DEVFSADM_IS_RUNNING(devfsadm_state) &&
 131                     !sdev_devfsadm_revoked()) {
 132                         /* wait 2 sec and check devfsadm completion */
 133                         rv = cv_reltimedwait_sig(&dv->sdev_lookup_cv,
 134                             &dv->sdev_lookup_lock, wakeup, TR_CLOCK_TICK);

 136                         if (is_lookup && (rv > 0)) {
 137                                 /* was this node constructed ? */
 138                                 if (dv->sdev_state == SDEV_READY) {
 139                                         rval = 0;
 140                                 }
 141                                 sdcmn_err6(("%s: wait done, %screated %d\n",
 142                                     dv->sdev_name, rval ? "not " : "",
 143                                     dv->sdev_state));
 144                                 break;
 145                         } else if (rv == 0) {
 146                                 /* interrupted */
 147                                 sdcmn_err6(("%s: wait interrupted\n",
 148                                     dv->sdev_name));
 149                                 break;
 150                         } else if ((rv == -1) &&
 151                             (ddi_get_lbolt() >= expire)) {
 152                                 sdcmn_err6(("%s: wait time is up\n",
 153                                     dv->sdev_name));
 154                                 break;
 155                         }
 156                         sdcmn_err6(("%s: wait "
 157                             "rv %ld state 0x%x expire %ld\n",
 158                             dv->sdev_name, rv, devfsadm_state,
 159                             expire - ddi_get_lbolt()));
 160                 }
 161         } else {
```



```
 162                 /*
 163                  * for the nodes created by
 164                  * devname_lookup_func callback
 165                  * or plug-in modules
 166                  */
 167                 while (SDEV_IS_LOOKUP(dv) || SDEV_IS_READDIR(dv)) {
 168                         cv_wait(&dv->sdev_lookup_cv, &dv->sdev_lookup_lock);
 169                 }
 170                 rval = 0;
 171         }

 173         sdcmn_err6(("wait4lookup unblocking %s state 0x%x %d\n",
 174             dv->sdev_name, devfsadm_state, dv->sdev_state));

 176         if (is_lookup) {
 177                 SDEV_UNBLOCK_OTHERS(dv, SDEV_LOOKUP);
 178         } else {
 179                 SDEV_UNBLOCK_OTHERS(dv, SDEV_READDIR);
 180         }

 182         return (rval);
 183 }
______unchanged_portion_omitted_

 227 static int
 228 sdev_open_upcall_door()
 229 {
 230         int error;
 231         clock_t rv;
 232         clock_t expire;

 234         ASSERT(sdev_upcall_door == NULL);

 236         /* timeout expires this many ticks in the future */
 237         expire = ddi_get_lbolt() + drv_sectohz(dev_devfsadm_startup);
 237         expire = ddi_get_lbolt() + drv_usectohz(dev_devfsadm_startup * 1000000);

 239         if (sdev_door_upcall_filename == NULL) {
 240                 if ((error = sdev_start_devfsadmd()) != 0) {
 241                         return (error);
 242                 }

 244                 /* wait for devfsadmd start */
 245                 mutex_enter(&devfsadm_lock);
 246                 while (sdev_door_upcall_filename == NULL) {
 247                         sdcmn_err6(("waiting for dev_door creation, %ld\n",
 248                             expire - ddi_get_lbolt()));
 249                         rv = cv_timedwait_sig(&devfsadm_cv, &devfsadm_lock,
 250                             expire);
 251                         sdcmn_err6(("dev_door wait rv %ld\n", rv));
 252                         if (rv <= 0) {
 253                                 sdcmn_err6(("devfsadmd startup error\n"));
 254                                 mutex_exit(&devfsadm_lock);
 255                                 return (EBADF);
 256                         }
 257                 }
 258                 sdcmn_err6(("devfsadmd is ready\n"));
 259                 mutex_exit(&devfsadm_lock);
 260         }

 262         if ((error = door_ki_open(sdev_door_upcall_filename,
 263             &sdev_upcall_door)) != 0) {
 264                 sdcmn_err6(("upcall_lookup: door open error %d\n",
 265                     error));
 266                 return (error);
 267         }
```

```
 269         return (0);
 270 }
______unchanged_portion_omitted_
```

****
24338 Wed Aug 19 07:24:53 2015
new/usr/src/uts/common/fs/dev/sdev_ncache.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 568 static void
 569 sdev_state_sysavail()
 570 {
 571         sdev_nc_list_t *ncl = sdev_ncache;
 572         clock_t nticks;
 573         int nsecs;

 575         mutex_enter(&ncl->ncl_mutex);
 576         ncl->ncl_flags |= NCL_LIST_WENABLE;
 577         mutex_exit(&ncl->ncl_mutex);

 579         nsecs = sdev_reconfig_delay;
 580         if (nsecs == 0) {
 581                 sdev_state_boot_complete();
 582         } else {
 583                 nticks = drv_sectohz(nsecs);
 583                 nticks = drv_usectohz(1000000 * nsecs);
 584                 sdcmn_err5(("timeout initiated %ld\n", nticks));
 585                 (void) timeout(sdev_state_timeout, NULL, nticks);
 586                 sdev_nc_flush_boot_update();
 587         }
 588 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   87634 Wed Aug 19 07:24:54 2015
new/usr/src/uts/common/fs/nfs/nfs4_stub_vnops.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 898 /*
 899  * Collect together both the generic & mount-type specific args.
 900  */
 901 static int
 902 nfs4_trigger_domount_args_create(vnode_t *vp, cred_t *cr, domount_args_t **dmap)
 903 {
 904         int nointr;
 905         char *hostlist;
 906         servinfo4_t *svp;
 907         struct nfs_args *nargs, *nargs_head;
 908         enum clnt_stat status;
 909         ephemeral_servinfo_t *esi, *esi_first;
 910         domount_args_t *dma;
 911         mntinfo4_t *mi = VTOMI4(vp);

 913         nointr = !(mi->mi_flags & MI4_INT);
 914         hostlist = kmem_zalloc(MAXPATHLEN, KM_SLEEP);

 916         svp = mi->mi_curr_serv;
 917         /* check if the current server is responding */
 918         status = nfs4_trigger_ping_server(svp, nointr);
 919         if (status == RPC_SUCCESS) {
 920                 esi_first = nfs4_trigger_esi_create(vp, svp, cr);
 921                 if (esi_first == NULL) {
 922                         kmem_free(hostlist, MAXPATHLEN);
 923                         return (EINVAL);
 924                 }

 926                 (void) strlcpy(hostlist, esi_first->esi_hostname, MAXPATHLEN);

 928                 nargs_head = nfs4_trigger_nargs_create(mi, svp, esi_first);
 929         } else {
 930                 /* current server did not respond */
 931                 esi_first = NULL;
 932                 nargs_head = NULL;
 933         }
 934         nargs = nargs_head;

 936         /*
 937          * NFS RO failover.
 938          *
 939          * If we have multiple servinfo4 structures, linked via sv_next,
 940          * we must create one nfs_args for each, linking the nfs_args via
 941          * nfs_ext_u.nfs_extB.next.
 942          *
 943          * We need to build a corresponding esi for each, too, but that is
 944          * used solely for building nfs_args, and may be immediately
 945          * discarded, as domount() requires the info from just one esi,
 946          * but all the nfs_args.
 947          *
 948          * Currently, the NFS mount code will hang if not all servers
 949          * requested are available. To avoid that, we need to ping each
 950          * server, here, and remove it from the list if it is not
 951          * responding. This has the side-effect of that server then
 952          * being permanently unavailable for this failover mount, even if
 953          * it recovers. That's unfortunate, but the best we can do until
 954          * the mount code path is fixed.
 955          */
```



```
 957         /*
 958          * If the current server was down, loop indefinitely until we find
 959          * at least one responsive server.
 960          */
 961         do {
 962                 /* no locking needed for sv_next; it is only set at fs mount */
 963                 for (svp = mi->mi_servers; svp != NULL; svp = svp->sv_next) {
 964                         struct nfs_args *next;

 966                         /*
 967                          * nargs_head: the head of the nfs_args list
 968                          * nargs: the current tail of the list
 969                          * next: the newly-created element to be added
 970                          */

 972                         /*
 973                          * We've already tried the current server, above;
 974                          * if it was responding, we have already included it
 975                          * and it may now be ignored.
 976                          *
 977                          * Otherwise, try it again, since it may now have
 978                          * recovered.
 979                          */
 980                         if (svp == mi->mi_curr_serv && esi_first != NULL)
 981                                 continue;

 983                         (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
 984                         if (svp->sv_flags & SV4_NOTINUSE) {
 985                                 nfs_rw_exit(&svp->sv_lock);
 986                                 continue;
 987                         }
 988                         nfs_rw_exit(&svp->sv_lock);

 990                         /* check if the server is responding */
 991                         status = nfs4_trigger_ping_server(svp, nointr);
 992                         if (status == RPC_INTR) {
 993                                 kmem_free(hostlist, MAXPATHLEN);
 994                                 nfs4_trigger_esi_destroy(esi_first, vp);
 995                                 nargs = nargs_head;
 996                                 while (nargs != NULL) {
 997                                         next = nargs->nfs_ext_u.nfs_extB.next;
 998                                         nfs4_trigger_nargs_destroy(nargs);
 999                                         nargs = next;
1000                                 }
1001                                 return (EINTR);
1002                         } else if (status != RPC_SUCCESS) {
1003                                 /* if the server did not respond, ignore it */
1004                                 continue;
1005                         }

1007                         esi = nfs4_trigger_esi_create(vp, svp, cr);
1008                         if (esi == NULL)
1009                                 continue;

1011                         /*
1012                          * If the original current server (mi_curr_serv)
1013                          * was down when when we first tried it,
1014                          * (i.e. esi_first == NULL),
1015                          * we select this new server (svp) to be the server
1016                          * that we will actually contact (esi_first).
1017                          *
1018                          * Note that it's possible that mi_curr_serv == svp,
1019                          * if that mi_curr_serv was down but has now recovered.
1020                          */
1021                         next = nfs4_trigger_nargs_create(mi, svp, esi);
1022                         if (esi_first == NULL) {
```

```
1023                                    ASSERT(nargs == NULL);
1024                                    ASSERT(nargs_head == NULL);
1025                                    nargs_head = next;
1026                                    esi_first = esi;
1027                                    (void) strlcpy(hostlist,
1028                                        esi_first->esi_hostname, MAXPATHLEN);
1029                            } else {
1030                                    ASSERT(nargs_head != NULL);
1031                                    nargs->nfs_ext_u.nfs_extB.next = next;
1032                                    (void) strlcat(hostlist, ",", MAXPATHLEN);
1033                                    (void) strlcat(hostlist, esi->esi_hostname,
1034                                        MAXPATHLEN);
1035                                    /* esi was only needed for hostname & nargs */
1036                                    nfs4_trigger_esi_destroy(esi, vp);
1037                            }

1039                            nargs = next;
1040                    }

1042                    /* if we've had no response at all, wait a second */
1043                    if (esi_first == NULL)
1044                            delay(drv_sectohz(1));
1044                            delay(drv_usectohz(1000000));

1046            } while (esi_first == NULL);
1047            ASSERT(nargs_head != NULL);

1049            dma = kmem_zalloc(sizeof (domount_args_t), KM_SLEEP);
1050            dma->dma_esi = esi_first;
1051            dma->dma_hostlist = hostlist;
1052            dma->dma_nargs = nargs_head;
1053            *dmap = dma;

1055            return (0);
1056    }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  242538 Wed Aug 19 07:24:54 2015
new/usr/src/uts/common/io/aac/aac.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 759 static int
 760 aac_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 761 {
 762         int instance, i;
 763         struct aac_softstate *softs = NULL;
 764         int attach_state = 0;
 765         char *data;

 767         DBCALLED(NULL, 1);

 769         switch (cmd) {
 770         case DDI_ATTACH:
 771                 break;
 772         case DDI_RESUME:
 773                 return (DDI_FAILURE);
 774         default:
 775                 return (DDI_FAILURE);
 776         }

 778         instance = ddi_get_instance(dip);

 780         /* Get soft state */
 781         if (ddi_soft_state_zalloc(aac_softstatep, instance) != DDI_SUCCESS) {
 782                 AACDB_PRINT(softs, CE_WARN, "Cannot alloc soft state");
 783                 goto error;
 784         }
 785         softs = ddi_get_soft_state(aac_softstatep, instance);
 786         attach_state |= AAC_ATTACH_SOFTSTATE_ALLOCED;

 788         softs->instance = instance;
 789         softs->devinfo_p = dip;
 790         softs->buf_dma_attr = softs->addr_dma_attr = aac_dma_attr;
 791         softs->addr_dma_attr.dma_attr_granular = 1;
 792         softs->acc_attr = aac_acc_attr;
 793         softs->reg_attr = aac_acc_attr;
 794         softs->card = AAC_UNKNOWN_CARD;
 795 #ifdef DEBUG
 796         softs->debug_flags = aac_debug_flags;
 797         softs->debug_fib_flags = aac_debug_fib_flags;
 798 #endif

 800         /* Initialize FMA */
 801         aac_fm_init(softs);

 803         /* Check the card type */
 804         if (aac_check_card_type(softs) == AACERR) {
 805                 AACDB_PRINT(softs, CE_WARN, "Card not supported");
 806                 goto error;
 807         }
 808         /* We have found the right card and everything is OK */
 809         attach_state |= AAC_ATTACH_CARD_DETECTED;

 811         /* Map PCI mem space */
 812         if (ddi_regs_map_setup(dip, 1,
 813             (caddr_t *)&softs->pci_mem_base_vaddr, 0,
 814             softs->map_size_min, &softs->reg_attr,
 815             &softs->pci_mem_handle) != DDI_SUCCESS)
 816                 goto error;
```



```
 818         softs->map_size = softs->map_size_min;
 819         attach_state |= AAC_ATTACH_PCI_MEM_MAPPED;

 821         AAC_DISABLE_INTR(softs);

 823         /* Init mutexes and condvars */
 824         mutex_init(&softs->io_lock, NULL, MUTEX_DRIVER,
 825             DDI_INTR_PRI(softs->intr_pri));
 826         mutex_init(&softs->q_comp_mutex, NULL, MUTEX_DRIVER,
 827             DDI_INTR_PRI(softs->intr_pri));
 828         mutex_init(&softs->time_mutex, NULL, MUTEX_DRIVER,
 829             DDI_INTR_PRI(softs->intr_pri));
 830         mutex_init(&softs->ev_lock, NULL, MUTEX_DRIVER,
 831             DDI_INTR_PRI(softs->intr_pri));
 832         mutex_init(&softs->aifq_mutex, NULL,
 833             MUTEX_DRIVER, DDI_INTR_PRI(softs->intr_pri));
 834         cv_init(&softs->event, NULL, CV_DRIVER, NULL);
 835         cv_init(&softs->sync_fib_cv, NULL, CV_DRIVER, NULL);
 836         cv_init(&softs->drain_cv, NULL, CV_DRIVER, NULL);
 837         cv_init(&softs->event_wait_cv, NULL, CV_DRIVER, NULL);
 838         cv_init(&softs->event_disp_cv, NULL, CV_DRIVER, NULL);
 839         cv_init(&softs->aifq_cv, NULL, CV_DRIVER, NULL);
 840         attach_state |= AAC_ATTACH_KMUTEX_INITED;

 842         /* Init the cmd queues */
 843         for (i = 0; i < AAC_CMDQ_NUM; i++)
 844                 aac_cmd_initq(&softs->q_wait[i]);
 845         aac_cmd_initq(&softs->q_busy);
 846         aac_cmd_initq(&softs->q_comp);

 848         /* Check for legacy device naming support */
 849         softs->legacy = 1; /* default to use legacy name */
 850         if ((ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
 851             "legacy-name-enable", &data) == DDI_SUCCESS)) {
 852                 if (strcmp(data, "no") == 0) {
 853                         AACDB_PRINT(softs, CE_NOTE, "legacy-name disabled");
 854                         softs->legacy = 0;
 855                 }
 856                 ddi_prop_free(data);
 857         }

 859         /*
 860          * Everything has been set up till now,
 861          * we will do some common attach.
 862          */
 863         mutex_enter(&softs->io_lock);
 864         if (aac_common_attach(softs) == AACERR) {
 865                 mutex_exit(&softs->io_lock);
 866                 goto error;
 867         }
 868         mutex_exit(&softs->io_lock);
 869         attach_state |= AAC_ATTACH_COMM_SPACE_SETUP;

 871         /* Check for buf breakup support */
 872         if ((ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
 873             "breakup-enable", &data) == DDI_SUCCESS)) {
 874                 if (strcmp(data, "yes") == 0) {
 875                         AACDB_PRINT(softs, CE_NOTE, "buf breakup enabled");
 876                         softs->flags |= AAC_FLAGS_BRKUP;
 877                 }
 878                 ddi_prop_free(data);
 879         }
 880         softs->dma_max = softs->buf_dma_attr.dma_attr_maxxfer;
 881         if (softs->flags & AAC_FLAGS_BRKUP) {
 882                 softs->dma_max = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
 883                     DDI_PROP_DONTPASS, "dma-max", softs->dma_max);
```

```
884         }

886         if (aac_hba_setup(softs) != AACOK)
887                 goto error;
888         attach_state |= AAC_ATTACH_SCSI_TRAN_SETUP;

890         /* Create devctl/scsi nodes for cfgadm */
891         if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
892             INST2DEVCTL(instance), DDI_NT_SCSI_NEXUS, 0) != DDI_SUCCESS) {
893                 AACDB_PRINT(softs, CE_WARN, "failed to create devctl node");
894                 goto error;
895         }
896         attach_state |= AAC_ATTACH_CREATE_DEVCTL;

898         if (ddi_create_minor_node(dip, "scsi", S_IFCHR, INST2SCSI(instance),
899             DDI_NT_SCSI_ATTACHMENT_POINT, 0) != DDI_SUCCESS) {
900                 AACDB_PRINT(softs, CE_WARN, "failed to create scsi node");
901                 goto error;
902         }
903         attach_state |= AAC_ATTACH_CREATE_SCSI;

905         /* Create aac node for app. to issue ioctls */
906         if (ddi_create_minor_node(dip, "aac", S_IFCHR, INST2AAC(instance),
907             DDI_PSEUDO, 0) != DDI_SUCCESS) {
908                 AACDB_PRINT(softs, CE_WARN, "failed to create aac node");
909                 goto error;
910         }

912         /* Common attach is OK, so we are attached! */
913         softs->state |= AAC_STATE_RUN;

915         /* Create event thread */
916         softs->fibctx_p = &softs->aifctx;
917         if ((softs->event_thread = thread_create(NULL, 0, aac_event_thread,
918             softs, 0, &p0, TS_RUN, minclsyspri)) == NULL) {
919                 AACDB_PRINT(softs, CE_WARN, "aif thread create failed");
920                 softs->state &= ~AAC_STATE_RUN;
921                 goto error;
922         }

924         aac_unhold_bus(softs, AAC_IOCMD_SYNC | AAC_IOCMD_ASYNC);

926         /* Create a thread for command timeout */
927         softs->timeout_id = timeout(aac_timer, (void *)softs,
928             drv_sectohz(aac_tick));
928             (aac_tick * drv_usectohz(1000000)));

930         /* Common attach is OK, so we are attached! */
931         ddi_report_dev(dip);
932         AACDB_PRINT(softs, CE_NOTE, "aac attached ok");
933         return (DDI_SUCCESS);

935 error:
936         if (attach_state & AAC_ATTACH_CREATE_SCSI)
937                 ddi_remove_minor_node(dip, "scsi");
938         if (attach_state & AAC_ATTACH_CREATE_DEVCTL)
939                 ddi_remove_minor_node(dip, "devctl");
940         if (attach_state & AAC_ATTACH_COMM_SPACE_SETUP)
941                 aac_common_detach(softs);
942         if (attach_state & AAC_ATTACH_SCSI_TRAN_SETUP) {
943                 (void) scsi_hba_detach(dip);
944                 scsi_hba_tran_free(AAC_DIP2TRAN(dip));
945         }
946         if (attach_state & AAC_ATTACH_KMUTEX_INITED) {
947                 mutex_destroy(&softs->io_lock);
948                 mutex_destroy(&softs->q_comp_mutex);
```



```
949                 mutex_destroy(&softs->time_mutex);
950                 mutex_destroy(&softs->ev_lock);
951                 mutex_destroy(&softs->aifq_mutex);
952                 cv_destroy(&softs->event);
953                 cv_destroy(&softs->sync_fib_cv);
954                 cv_destroy(&softs->drain_cv);
955                 cv_destroy(&softs->event_wait_cv);
956                 cv_destroy(&softs->event_disp_cv);
957                 cv_destroy(&softs->aifq_cv);
958         }
959         if (attach_state & AAC_ATTACH_PCI_MEM_MAPPED)
960                 ddi_regs_map_free(&softs->pci_mem_handle);
961         aac_fm_fini(softs);
962         if (attach_state & AAC_ATTACH_CARD_DETECTED)
963                 softs->card = AACERR;
964         if (attach_state & AAC_ATTACH_SOFTSTATE_ALLOCED)
965                 ddi_soft_state_free(aac_softstatep, instance);
966         return (DDI_FAILURE);
967 }
______unchanged_portion_omitted_

4360 /*
4361  * The draining thread is shared among quiesce threads. It terminates
4362  * when the adapter is quiesced or stopped by aac_stop_drain().
4363  */
4364 static void
4365 aac_check_drain(void *arg)
4366 {
4367         struct aac_softstate *softs = arg;

4369         mutex_enter(&softs->io_lock);
4370         if (softs->ndrains) {
4371                 softs->drain_timeid = 0;
4372                 /*
4373                  * If both ASYNC and SYNC bus throttle are held,
4374                  * wake up threads only when both are drained out.
4375                  */
4376                 if ((softs->bus_throttle[AAC_CMDQ_ASYNC] > 0 ||
4377                     softs->bus_ncmds[AAC_CMDQ_ASYNC] == 0) &&
4378                     (softs->bus_throttle[AAC_CMDQ_SYNC] > 0 ||
4379                     softs->bus_ncmds[AAC_CMDQ_SYNC] == 0))
4380                         cv_broadcast(&softs->drain_cv);
4381                 else
4382                         softs->drain_timeid = timeout(aac_check_drain, softs,
4383                             drv_sectohz(AAC_QUIESCE_TICK));
4383                             AAC_QUIESCE_TICK * drv_usectohz(1000000));
4384         }
4385         mutex_exit(&softs->io_lock);
4386 }

4388 /*
4389  * If not draining the outstanding cmds, drain them. Otherwise,
4390  * only update ndrains.
4391  */
4392 static void
4393 aac_start_drain(struct aac_softstate *softs)
4394 {
4395         if (softs->ndrains == 0) {
4396                 ASSERT(softs->drain_timeid == 0);
4397                 softs->drain_timeid = timeout(aac_check_drain, softs,
4398                     drv_sectohz(AAC_QUIESCE_TICK));
4398                     AAC_QUIESCE_TICK * drv_usectohz(1000000));
4399         }
4400         softs->ndrains++;
4401 }
______unchanged_portion_omitted_
```

```
6650 /*
6651  * Timeout checking and handling
6652  */
6653 static void
6654 aac_daemon(struct aac_softstate *softs)
6655 {
6656         int time_out; /* set if timeout happened */
6657         int time_adjust;
6658         uint32_t softs_timebase;

6660         mutex_enter(&softs->time_mutex);
6661         ASSERT(softs->time_out <= softs->timebase);
6662         softs->time_out = 0;
6663         softs_timebase = softs->timebase;
6664         mutex_exit(&softs->time_mutex);

6666         /* Check slots for timeout pkts */
6667         time_adjust = 0;
6668         do {
6669                 struct aac_cmd *acp;

6671                 time_out = 0;
6672                 for (acp = softs->q_busy.q_head; acp; acp = acp->next) {
6673                         if (acp->timeout == 0)
6674                                 continue;

6676                         /*
6677                          * If timeout happened, update outstanding cmds
6678                          * to be checked later again.
6679                          */
6680                         if (time_adjust) {
6681                                 acp->timeout += time_adjust;
6682                                 continue;
6683                         }

6685                         if (acp->timeout <= softs_timebase) {
6686                                 aac_cmd_timeout(softs, acp);
6687                                 time_out = 1;
6688                                 time_adjust = drv_sectohz(aac_tick);
6688                                 time_adjust = aac_tick * drv_usectohz(1000000);
6689                                 break; /* timeout happened */
6690                         } else {
6691                                 break; /* no timeout */
6692                         }
6693                 }
6694         } while (time_out);

6696         mutex_enter(&softs->time_mutex);
6697         softs->time_out = softs->timebase + aac_tick;
6698         mutex_exit(&softs->time_mutex);
6699 }
```

_____***unchanged_portion_omitted_***

```
6744 /*
6745  * Internal timer. It is only responsbile for time counting and report time
6746  * related events. Events handling is done by aac_event_thread(), so that
6747  * the timer itself could be as precise as possible.
6748  */
6749 static void
6750 aac_timer(void *arg)
6751 {
6752         struct aac_softstate *softs = arg;
6753         int events = 0;

6755         mutex_enter(&softs->time_mutex);
```



```
6757         /* If timer is being stopped, exit */
6758         if (softs->timeout_id) {
6759                 softs->timeout_id = timeout(aac_timer, (void *)softs,
6760                     drv_sectohz(aac_tick));
6760                     (aac_tick * drv_usectohz(1000000)));
6761         } else {
6762                 mutex_exit(&softs->time_mutex);
6763                 return;
6764         }

6766         /* Time counting */
6767         softs->timebase += aac_tick;

6769         /* Check time related events */
6770         if (softs->time_out && softs->time_out <= softs->timebase)
6771                 events |= AAC_EVENT_TIMEOUT;
6772         if (softs->time_sync && softs->time_sync <= softs->timebase)
6773                 events |= AAC_EVENT_SYNCTICK;

6775         mutex_exit(&softs->time_mutex);

6777         if (events)
6778                 aac_event_disp(softs, events);
6779 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   64430 Wed Aug 19 07:24:54 2015
new/usr/src/uts/common/io/aggr/aggr_lacp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 439 static void
 440 start_periodic_timer(aggr_port_t *portp)
 441 {
 442         aggr_lacp_port_t *pl = &portp->lp_lacp;

 444         ASSERT(MAC_PERIM_HELD(portp->lp_grp->lg_mh));

 446         mutex_enter(&pl->lacp_timer_lock);
 447         if (pl->periodic_timer.id == 0) {
 448                 pl->periodic_timer.id = timeout(periodic_timer_pop, portp,
 449                     drv_sectohz(portp->lp_lacp.periodic_timer.val));
 449                     drv_usectohz(1000000 * portp->lp_lacp.periodic_timer.val));
 450         }
 451         mutex_exit(&pl->lacp_timer_lock);
 452 }
______unchanged_portion_omitted_

1509 static void
1510 start_wait_while_timer(aggr_port_t *portp)
1511 {
1512         aggr_lacp_port_t *pl = &portp->lp_lacp;

1514         ASSERT(MAC_PERIM_HELD(portp->lp_grp->lg_mh));

1516         mutex_enter(&pl->lacp_timer_lock);
1517         if (pl->wait_while_timer.id == 0) {
1518                 pl->wait_while_timer.id =
1519                     timeout(wait_while_timer_pop, portp,
1520                     drv_sectohz(portp->lp_lacp.wait_while_timer.val));
1520                     drv_usectohz(1000000 *
1521                     portp->lp_lacp.wait_while_timer.val));
1521         }
1522         mutex_exit(&pl->lacp_timer_lock);
1523 }
______unchanged_portion_omitted_
```

```
**********************************************************
  130175 Wed Aug 19 07:24:55 2015
new/usr/src/uts/common/io/asy.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3021 /*
3022  * Start output on a line, unless it's busy, frozen, or otherwise.
3023  */
3024 /*ARGSUSED*/
3025 static void
3026 async_nstart(struct asyncline *async, int mode)
3027 {
3028         struct asycom *asy = async->async_common;
3029         int cc;
3030         queue_t *q;
3031         mblk_t *bp;
3032         uchar_t *xmit_addr;
3033         uchar_t val;
3034         int     fifo_len = 1;
3035         boolean_t didsome;
3036         mblk_t *nbp;

3038 #ifdef DEBUG
3039         int instance = UNIT(async->async_dev);

3041         DEBUGCONT1(ASY_DEBUG_PROCS, "async%d_nstart\n", instance);
3042 #endif
3043         if (asy->asy_use_fifo == FIFO_ON) {
3044                 fifo_len = asy->asy_fifo_buf; /* with FIFO buffers */
3045                 if (fifo_len > asy_max_tx_fifo)
3046                         fifo_len = asy_max_tx_fifo;
3047         }

3049         ASSERT(mutex_owned(&asy->asy_excl));

3051         /*
3052          * If the chip is busy (i.e., we're waiting for a break timeout
3053          * to expire, or for the current transmission to finish, or for
3054          * output to finish draining from chip), don't grab anything new.
3055          */
3056         if (async->async_flags & (ASYNC_BREAK|ASYNC_BUSY)) {
3057                 DEBUGCONT2((mode? ASY_DEBUG_OUT : 0),
3058                     "async%d_nstart: start %s.\n",
3059                     instance,
3060                     async->async_flags & ASYNC_BREAK ? "break" : "busy");
3061                 return;
3062         }

3064         /*
3065          * Check only pended sw input flow control.
3066          */
3067         mutex_enter(&asy->asy_excl_hi);
3068         if (async_flowcontrol_sw_input(asy, FLOW_CHECK, IN_FLOW_NULL))
3069                 fifo_len--;
3070         mutex_exit(&asy->asy_excl_hi);

3072         /*
3073          * If we're waiting for a delay timeout to expire, don't grab
3074          * anything new.
3075          */
3076         if (async->async_flags & ASYNC_DELAY) {
3077                 DEBUGCONT1((mode? ASY_DEBUG_OUT : 0),
3078                     "async%d_nstart: start ASYNC_DELAY.\n", instance);
3079                 return;
```



```
3080         }

3082         if ((q = async->async_ttycommon.t_writeq) == NULL) {
3083                 DEBUGCONT1((mode? ASY_DEBUG_OUT : 0),
3084                     "async%d_nstart: start writeq is null.\n", instance);
3085                 return; /* not attached to a stream */
3086         }

3088         for (;;) {
3089                 if ((bp = getq(q)) == NULL)
3090                         return; /* no data to transmit */

3092                 /*
3093                  * We have a message block to work on.
3094                  * Check whether it's a break, a delay, or an ioctl (the latter
3095                  * occurs if the ioctl in question was waiting for the output
3096                  * to drain).  If it's one of those, process it immediately.
3097                  */
3098                 switch (bp->b_datap->db_type) {

3100                 case M_BREAK:
3101                         /*
3102                          * Set the break bit, and arrange for "async_restart"
3103                          * to be called in 1/4 second; it will turn the
3104                          * break bit off, and call "async_start" to grab
3105                          * the next message.
3106                          */
3107                         mutex_enter(&asy->asy_excl_hi);
3108                         val = ddi_get8(asy->asy_iohandle,
3109                             asy->asy_ioaddr + LCR);
3110                         ddi_put8(asy->asy_iohandle, asy->asy_ioaddr + LCR,
3111                             (val | SETBREAK));
3112                         mutex_exit(&asy->asy_excl_hi);
3113                         async->async_flags |= ASYNC_BREAK;
3114                         (void) timeout(async_restart, (caddr_t)async,
3115                             drv_sectohz(1) / 4);
3115                             drv_usectohz(1000000)/4);
3116                         freemsg(bp);
3117                         return; /* wait for this to finish */

3119                 case M_DELAY:
3120                         /*
3121                          * Arrange for "async_restart" to be called when the
3122                          * delay expires; it will turn ASYNC_DELAY off,
3123                          * and call "async_start" to grab the next message.
3124                          */
3125                         (void) timeout(async_restart, (caddr_t)async,
3126                             (int)(*(unsigned char *)bp->b_rptr + 6));
3127                         async->async_flags |= ASYNC_DELAY;
3128                         freemsg(bp);
3129                         return; /* wait for this to finish */

3131                 case M_IOCTL:
3132                         /*
3133                          * This ioctl was waiting for the output ahead of
3134                          * it to drain; obviously, it has.  Do it, and
3135                          * then grab the next message after it.
3136                          */
3137                         mutex_exit(&asy->asy_excl);
3138                         async_ioctl(async, q, bp);
3139                         mutex_enter(&asy->asy_excl);
3140                         continue;
3141                 }

3143                 while (bp != NULL && ((cc = MBLKL(bp)) == 0)) {
3144                         nbp = bp->b_cont;
```

```
3145                        freeb(bp);
3146                        bp = nbp;
3147                }
3148                if (bp != NULL)
3149                        break;
3150        }

3152        /*
3153         * We have data to transmit.  If output is stopped, put
3154         * it back and try again later.
3155         */
3156        if (async->async_flags & (ASYNC_HW_OUT_FLW | ASYNC_SW_OUT_FLW |
3157            ASYNC_STOPPED | ASYNC_OUT_SUSPEND)) {
3158                (void) putbq(q, bp);
3159                return;
3160        }

3162        async->async_xmitblk = bp;
3163        xmit_addr = bp->b_rptr;
3164        bp = bp->b_cont;
3165        if (bp != NULL)
3166                (void) putbq(q, bp);    /* not done with this message yet */

3168        /*
3169         * In 5-bit mode, the high order bits are used
3170         * to indicate character sizes less than five,
3171         * so we need to explicitly mask before transmitting
3172         */
3173        if ((async->async_ttycommon.t_cflag & CSIZE) == CS5) {
3174                unsigned char *p = xmit_addr;
3175                int cnt = cc;

3177                while (cnt--)
3178                        *p++ &= (unsigned char) 0x1f;
3179        }

3181        /*
3182         * Set up this block for pseudo-DMA.
3183         */
3184        mutex_enter(&asy->asy_excl_hi);
3185        /*
3186         * If the transmitter is ready, shove the first
3187         * character out.
3188         */
3189        didsome = B_FALSE;
3190        while (--fifo_len >= 0 && cc > 0) {
3191                if (!(ddi_get8(asy->asy_iohandle, asy->asy_ioaddr + LSR) &
3192                    XHRE))
3193                        break;
3194                ddi_put8(asy->asy_iohandle, asy->asy_ioaddr + DAT,
3195                    *xmit_addr++);
3196                cc--;
3197                didsome = B_TRUE;
3198        }
3199        async->async_optr = xmit_addr;
3200        async->async_ocnt = cc;
3201        if (didsome)
3202                async->async_flags |= ASYNC_PROGRESS;
3203        DEBUGCONT2(ASY_DEBUG_BUSY,
3204            "async%d_nstart: Set ASYNC_BUSY.  async_ocnt=%d\n",
3205            instance, async->async_ocnt);
3206        async->async_flags |= ASYNC_BUSY;
3207        mutex_exit(&asy->asy_excl_hi);
3208 }
```

unchanged_portion_omitted_


```
3301 /*
3302  * Process an "ioctl" message sent down to us.
3303  * Note that we don't need to get any locks until we are ready to access
3304  * the hardware.  Nothing we access until then is going to be altered
3305  * outside of the STREAMS framework, so we should be safe.
3306  */
3307 int asydelay = 10000;
3308 static void
3309 async_ioctl(struct asyncline *async, queue_t *wq, mblk_t *mp)
3310 {
3311        struct asycom *asy = async->async_common;
3312        tty_common_t  *tp = &async->async_ttycommon;
3313        struct iocblk *iocp;
3314        unsigned datasize;
3315        int error = 0;
3316        uchar_t val;
3317        mblk_t *datamp;
3318        unsigned int index;

3320 #ifdef DEBUG
3321        int instance = UNIT(async->async_dev);

3323        DEBUGCONT1(ASY_DEBUG_PROCS, "async%d_ioctl\n", instance);
3324 #endif

3326        if (tp->t_iocpending != NULL) {
3327                /*
3328                 * We were holding an "ioctl" response pending the
3329                 * availability of an "mblk" to hold data to be passed up;
3330                 * another "ioctl" came through, which means that "ioctl"
3331                 * must have timed out or been aborted.
3332                 */
3333                freemsg(async->async_ttycommon.t_iocpending);
3334                async->async_ttycommon.t_iocpending = NULL;
3335        }

3337        iocp = (struct iocblk *)mp->b_rptr;

3339        /*
3340         * For TIOCMGET and the PPS ioctls, do NOT call ttycommon_ioctl()
3341         * because this function frees up the message block (mp->b_cont) that
3342         * contains the user location where we pass back the results.
3343         *
3344         * Similarly, CONSOPENPOLLEDIO needs ioc_count, which ttycommon_ioctl
3345         * zaps.  We know that ttycommon_ioctl doesn't know any CONS*
3346         * ioctls, so keep the others safe too.
3347         */
3348        DEBUGCONT2(ASY_DEBUG_IOCTL, "async%d_ioctl: %s\n",
3349            instance,
3350            iocp->ioc_cmd == TIOCMGET ? "TIOCMGET" :
3351            iocp->ioc_cmd == TIOCMSET ? "TIOCMSET" :
3352            iocp->ioc_cmd == TIOCMBIS ? "TIOCMBIS" :
3353            iocp->ioc_cmd == TIOCMBIC ? "TIOCMBIC" :
3354            "other");

3356        switch (iocp->ioc_cmd) {
3357        case TIOCMGET:
3358        case TIOCGPPS:
3359        case TIOCSPPS:
3360        case TIOCGPPSEV:
3361        case CONSOPENPOLLEDIO:
3362        case CONSCLOSEPOLLEDIO:
3363        case CONSSETABORTENABLE:
3364        case CONSGETABORTENABLE:
3365                error = -1; /* Do Nothing */
3366                break;
```

```
3367         default:

3369                 /*
3370                  * The only way in which "ttycommon_ioctl" can fail is if the
3371                  * "ioctl" requires a response containing data to be returned
3372                  * to the user, and no mblk could be allocated for the data.
3373                  * No such "ioctl" alters our state.  Thus, we always go ahead
3374                  * and do any state-changes the "ioctl" calls for.  If we
3375                  * couldn't allocate the data, "ttycommon_ioctl" has stashed
3376                  * the "ioctl" away safely, so we just call "bufcall" to
3377                  * request that we be called back when we stand a better
3378                  * chance of allocating the data.
3379                  */
3380                 if ((datasize = ttycommon_ioctl(tp, wq, mp, &error)) != 0) {
3381                         if (async->async_wbufcid)
3382                                 unbufcall(async->async_wbufcid);
3383                         async->async_wbufcid = bufcall(datasize, BPRI_HI,
3384                             (void (*)(void *)) async_reioctl,
3385                             (void *)(intptr_t)async->async_common->asy_unit);
3386                         return;
3387                 }
3388         }

3390         mutex_enter(&asy->asy_excl);

3392         if (error == 0) {
3393                 /*
3394                  * "ttycommon_ioctl" did most of the work; we just use the
3395                  * data it set up.
3396                  */
3397                 switch (iocp->ioc_cmd) {

3399                 case TCSETS:
3400                         mutex_enter(&asy->asy_excl_hi);
3401                         if (asy_baudok(asy))
3402                                 asy_program(asy, ASY_NOINIT);
3403                         else
3404                                 error = EINVAL;
3405                         mutex_exit(&asy->asy_excl_hi);
3406                         break;
3407                 case TCSETSF:
3408                 case TCSETSW:
3409                 case TCSETA:
3410                 case TCSETAW:
3411                 case TCSETAF:
3412                         mutex_enter(&asy->asy_excl_hi);
3413                         if (!asy_baudok(asy))
3414                                 error = EINVAL;
3415                         else {
3416                                 if (asy_isbusy(asy))
3417                                         asy_waiteot(asy);
3418                                 asy_program(asy, ASY_NOINIT);
3419                         }
3420                         mutex_exit(&asy->asy_excl_hi);
3421                         break;
3422                 }
3423         } else if (error < 0) {
3424                 /*
3425                  * "ttycommon_ioctl" didn't do anything; we process it here.
3426                  */
3427                 error = 0;
3428                 switch (iocp->ioc_cmd) {

3430                 case TIOCGPPS:
3431                         /*
3432                          * Get PPS on/off.
```



```
3433                          */
3434                         if (mp->b_cont != NULL)
3435                                 freemsg(mp->b_cont);

3437                         mp->b_cont = allocb(sizeof (int), BPRI_HI);
3438                         if (mp->b_cont == NULL) {
3439                                 error = ENOMEM;
3440                                 break;
3441                         }
3442                         if (asy->asy_flags & ASY_PPS)
3443                                 *(int *)mp->b_cont->b_wptr = 1;
3444                         else
3445                                 *(int *)mp->b_cont->b_wptr = 0;
3446                         mp->b_cont->b_wptr += sizeof (int);
3447                         mp->b_datap->db_type = M_IOCACK;
3448                         iocp->ioc_count = sizeof (int);
3449                         break;

3451                 case TIOCSPPS:
3452                         /*
3453                          * Set PPS on/off.
3454                          */
3455                         error = miocpullup(mp, sizeof (int));
3456                         if (error != 0)
3457                                 break;

3459                         mutex_enter(&asy->asy_excl_hi);
3460                         if (*(int *)mp->b_cont->b_rptr)
3461                                 asy->asy_flags |= ASY_PPS;
3462                         else
3463                                 asy->asy_flags &= ~ASY_PPS;
3464                         /* Reset edge sense */
3465                         asy->asy_flags &= ~ASY_PPS_EDGE;
3466                         mutex_exit(&asy->asy_excl_hi);
3467                         mp->b_datap->db_type = M_IOCACK;
3468                         break;

3470                 case TIOCGPPSEV:
3471                 {
3472                         /*
3473                          * Get PPS event data.
3474                          */
3475                         mblk_t *bp;
3476                         void *buf;
3477 #ifdef _SYSCALL32_IMPL
3478                         struct ppsclockev32 p32;
3479 #endif
3480                         struct ppsclockev ppsclockev;

3482                         if (mp->b_cont != NULL) {
3483                                 freemsg(mp->b_cont);
3484                                 mp->b_cont = NULL;
3485                         }

3487                         if ((asy->asy_flags & ASY_PPS) == 0) {
3488                                 error = ENXIO;
3489                                 break;
3490                         }

3492                         /* Protect from incomplete asy_ppsev */
3493                         mutex_enter(&asy->asy_excl_hi);
3494                         ppsclockev = asy_ppsev;
3495                         mutex_exit(&asy->asy_excl_hi);

3497 #ifdef _SYSCALL32_IMPL
3498                         if ((iocp->ioc_flag & IOC_MODELS) != IOC_NATIVE) {
```

```
3499                                TIMEVAL_TO_TIMEVAL32(&p32.tv, &ppsclockev.tv);
3500                                p32.serial = ppsclockev.serial;
3501                                buf = &p32;
3502                                iocp->ioc_count = sizeof (struct ppsclockev32);
3503                        } else
3504 #endif
3505                        {
3506                                buf = &ppsclockev;
3507                                iocp->ioc_count = sizeof (struct ppsclockev);
3508                        }

3510                        if ((bp = allocb(iocp->ioc_count, BPRI_HI)) == NULL) {
3511                                error = ENOMEM;
3512                                break;
3513                        }
3514                        mp->b_cont = bp;

3516                        bcopy(buf, bp->b_wptr, iocp->ioc_count);
3517                        bp->b_wptr += iocp->ioc_count;
3518                        mp->b_datap->db_type = M_IOCACK;
3519                        break;
3520                }

3522                case TCSBRK:
3523                        error = miocpullup(mp, sizeof (int));
3524                        if (error != 0)
3525                                break;

3527                        if (*(int *)mp->b_cont->b_rptr == 0) {

3529                                /*
3530                                 * XXX Arrangements to ensure that a break
3531                                 * isn't in progress should be sufficient.
3532                                 * This ugly delay() is the only thing
3533                                 * that seems to work on the NCR Worldmark.
3534                                 * It should be replaced. Note that an
3535                                 * asy_waiteot() also does not work.
3536                                 */
3537                                if (asydelay)
3538                                        delay(drv_usectohz(asydelay));

3540                                while (async->async_flags & ASYNC_BREAK) {
3541                                        cv_wait(&async->async_flags_cv,
3542                                            &asy->asy_excl);
3543                                }
3544                                mutex_enter(&asy->asy_excl_hi);
3545                                /*
3546                                 * We loop until the TSR is empty and then
3547                                 * set the break.  ASYNC_BREAK has been set
3548                                 * to ensure that no characters are
3549                                 * transmitted while the TSR is being
3550                                 * flushed and SOUT is being used for the
3551                                 * break signal.
3552                                 *
3553                                 * The wait period is equal to
3554                                 * clock / (baud * 16) * 16 * 2.
3555                                 */
3556                                index = BAUDINDEX(
3557                                    async->async_ttycommon.t_cflag);
3558                                async->async_flags |= ASYNC_BREAK;

3560                                while ((ddi_get8(asy->asy_iohandle,
3561                                    asy->asy_ioaddr + LSR) & XSRE) == 0) {
3562                                        mutex_exit(&asy->asy_excl_hi);
3563                                        mutex_exit(&asy->asy_excl);
3564                                        drv_usecwait(
```



```
3565                                            32*asyspdtab[index] & 0xfff);
3566                                        mutex_enter(&asy->asy_excl);
3567                                        mutex_enter(&asy->asy_excl_hi);
3568                                }
3569                                /*
3570                                 * Arrange for "async_restart"
3571                                 * to be called in 1/4 second;
3572                                 * it will turn the break bit off, and call
3573                                 * "async_start" to grab the next message.
3574                                 */
3575                                val = ddi_get8(asy->asy_iohandle,
3576                                    asy->asy_ioaddr + LCR);
3577                                ddi_put8(asy->asy_iohandle,
3578                                    asy->asy_ioaddr + LCR,
3579                                    (val | SETBREAK));
3580                                mutex_exit(&asy->asy_excl_hi);
3581                                (void) timeout(async_restart, (caddr_t)async,
3582                                    drv_sectohz(1) / 4);
3582                                    drv_usectohz(1000000)/4);
3583                        } else {
3584                                DEBUGCONT1(ASY_DEBUG_OUT,
3585                                    "async%d_ioctl: wait for flush.\n",
3586                                    instance);
3587                                mutex_enter(&asy->asy_excl_hi);
3588                                asy_waiteot(asy);
3589                                mutex_exit(&asy->asy_excl_hi);
3590                                DEBUGCONT1(ASY_DEBUG_OUT,
3591                                    "async%d_ioctl: ldterm satisfied.\n",
3592                                    instance);
3593                        }
3594                        break;

3596                case TIOCSBRK:
3597                        if (!(async->async_flags & ASYNC_OUT_SUSPEND)) {
3598                                mutex_enter(&asy->asy_excl_hi);
3599                                async->async_flags |= ASYNC_OUT_SUSPEND;
3600                                async->async_flags |= ASYNC_HOLD_UTBRK;
3601                                index = BAUDINDEX(
3602                                    async->async_ttycommon.t_cflag);
3603                                while ((ddi_get8(asy->asy_iohandle,
3604                                    asy->asy_ioaddr + LSR) & XSRE) == 0) {
3605                                        mutex_exit(&asy->asy_excl_hi);
3606                                        mutex_exit(&asy->asy_excl);
3607                                        drv_usecwait(
3608                                            32*asyspdtab[index] & 0xfff);
3609                                        mutex_enter(&asy->asy_excl);
3610                                        mutex_enter(&asy->asy_excl_hi);
3611                                }
3612                                val = ddi_get8(asy->asy_iohandle,
3613                                    asy->asy_ioaddr + LCR);
3614                                ddi_put8(asy->asy_iohandle,
3615                                    asy->asy_ioaddr + LCR, (val | SETBREAK));
3616                                mutex_exit(&asy->asy_excl_hi);
3617                                /* wait for 100ms to hold BREAK */
3618                                async->async_utbrktid =
3619                                    timeout((void (*)())async_hold_utbrk,
3620                                    (caddr_t)async,
3621                                    drv_usectohz(asy_min_utbrk));
3622                        }
3623                        mioc2ack(mp, NULL, 0, 0);
3624                        break;

3626                case TIOCCBRK:
3627                        if (async->async_flags & ASYNC_OUT_SUSPEND)
3628                                async_resume_utbrk(async);
3629                        mioc2ack(mp, NULL, 0, 0);
```

```
3630                        break;

3632                case TIOCMSET:
3633                case TIOCMBIS:
3634                case TIOCMBIC:
3635                        if (iocp->ioc_count != TRANSPARENT) {
3636                                DEBUGCONT1(ASY_DEBUG_IOCTL, "async%d_ioctl: "
3637                                    "non-transparent\n", instance);

3639                                error = miocpullup(mp, sizeof (int));
3640                                if (error != 0)
3641                                        break;

3643                                mutex_enter(&asy->asy_excl_hi);
3644                                (void) asymctl(asy,
3645                                    dmtoasy(*(int *)mp->b_cont->b_rptr),
3646                                    iocp->ioc_cmd);
3647                                mutex_exit(&asy->asy_excl_hi);
3648                                iocp->ioc_error = 0;
3649                                mp->b_datap->db_type = M_IOCACK;
3650                        } else {
3651                                DEBUGCONT1(ASY_DEBUG_IOCTL, "async%d_ioctl: "
3652                                    "transparent\n", instance);
3653                                mcopyin(mp, NULL, sizeof (int), NULL);
3654                        }
3655                        break;

3657                case TIOCMGET:
3658                        datamp = allocb(sizeof (int), BPRI_MED);
3659                        if (datamp == NULL) {
3660                                error = EAGAIN;
3661                                break;
3662                        }

3664                        mutex_enter(&asy->asy_excl_hi);
3665                        *(int *)datamp->b_rptr = asymctl(asy, 0, TIOCMGET);
3666                        mutex_exit(&asy->asy_excl_hi);

3668                        if (iocp->ioc_count == TRANSPARENT) {
3669                                DEBUGCONT1(ASY_DEBUG_IOCTL, "async%d_ioctl: "
3670                                    "transparent\n", instance);
3671                                mcopyout(mp, NULL, sizeof (int), NULL, datamp);
3672                        } else {
3673                                DEBUGCONT1(ASY_DEBUG_IOCTL, "async%d_ioctl: "
3674                                    "non-transparent\n", instance);
3675                                mioc2ack(mp, datamp, sizeof (int), 0);
3676                        }
3677                        break;

3679                case CONSOPENPOLLEDIO:
3680                        error = miocpullup(mp, sizeof (struct cons_polledio *));
3681                        if (error != 0)
3682                                break;

3684                        *(struct cons_polledio **)mp->b_cont->b_rptr =
3685                            &asy->polledio;

3687                        mp->b_datap->db_type = M_IOCACK;
3688                        break;

3690                case CONSCLOSEPOLLEDIO:
3691                        mp->b_datap->db_type = M_IOCACK;
3692                        iocp->ioc_error = 0;
3693                        iocp->ioc_rval = 0;
3694                        break;
```



```
3696                case CONSSETABORTENABLE:
3697                        error = secpolicy_console(iocp->ioc_cr);
3698                        if (error != 0)
3699                                break;

3701                        if (iocp->ioc_count != TRANSPARENT) {
3702                                error = EINVAL;
3703                                break;
3704                        }

3706                        if (*(intptr_t *)mp->b_cont->b_rptr)
3707                                asy->asy_flags |= ASY_CONSOLE;
3708                        else
3709                                asy->asy_flags &= ~ASY_CONSOLE;

3711                        mp->b_datap->db_type = M_IOCACK;
3712                        iocp->ioc_error = 0;
3713                        iocp->ioc_rval = 0;
3714                        break;

3716                case CONSGETABORTENABLE:
3717                        /*CONSTANTCONDITION*/
3718                        ASSERT(sizeof (boolean_t) <= sizeof (boolean_t *));
3719                        /*
3720                         * Store the return value right in the payload
3721                         * we were passed.  Crude.
3722                         */
3723                        mcopyout(mp, NULL, sizeof (boolean_t), NULL, NULL);
3724                        *(boolean_t *)mp->b_cont->b_rptr =
3725                            (asy->asy_flags & ASY_CONSOLE) != 0;
3726                        break;

3728                default:
3729                        /*
3730                         * If we don't understand it, it's an error.  NAK it.
3731                         */
3732                        error = EINVAL;
3733                        break;
3734                }
3735        }
3736        if (error != 0) {
3737                iocp->ioc_error = error;
3738                mp->b_datap->db_type = M_IOCNAK;
3739        }
3740        mutex_exit(&asy->asy_excl);
3741        qreply(wq, mp);
3742        DEBUGCONT1(ASY_DEBUG_PROCS, "async%d_ioctl: done\n", instance);
3743 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   42515 Wed Aug 19 07:24:55 2015
new/usr/src/uts/common/io/bge/bge_impl.h
XXXX introduce drv_sectohz
**********************************************************
```

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

/*
 * Copyright (c) 2010-2013, by Broadcom, Inc.
 * All Rights Reserved.
 */

/*
 * Copyright (c) 2002, 2010, Oracle and/or its affiliates.
 * All rights reserved.
 */

#ifndef _BGE_IMPL_H
#define _BGE_IMPL_H


#ifdef __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stat.h>
#include <sys/pci.h>
#include <sys/note.h>
#include <sys/modctl.h>
#include <sys/crc32.h>
#ifdef  __sparcv9
#include <v9/sys/membar.h>
#endif  /* __sparcv9 */
#include <sys/kstat.h>
#include <sys/ethernet.h>
#include <sys/errno.h>
#include <sys/dlpi.h>
#include <sys/devops.h>
#include <sys/debug.h>
#include <sys/conf.h>

#include <netinet/ip6.h>
```



```
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/mi.h>
#include <inet/nd.h>
#include <sys/pattr.h>

#include <sys/disp.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

#include <sys/ddifm.h>
#include <sys/fm/protocol.h>
#include <sys/fm/util.h>
#include <sys/fm/io/ddi.h>

#include <sys/mac_provider.h>
#include <sys/mac_ether.h>

#ifdef __amd64
#include <sys/x86_archext.h>
#endif

#ifndef VLAN_TAGSZ
#define VLAN_TAGSZ 4
#endif

#define BGE_STR_SIZE 32

#ifndef OFFSETOF
#define OFFSETOF(_s, _f) \
    ((uint32_t)((uint8_t *)(&((_s *)0)->_f) - \
                (uint8_t *)((uint8_t *) 0)))
#endif

/*
 * <sys/ethernet.h> *may* already have provided the typedef ether_addr_t;
 * but of course C doesn't provide a way to check this directly.  So here
 * we rely on the fact that the symbol ETHERTYPE_AT was added to the
 * header file (as a #define, which we *can* test for) at the same time
 * as the typedef for ether_addr_t ;-!
 */
#ifndef ETHERTYPE_AT
typedef uchar_t ether_addr_t[ETHERADDRL];
#endif  /* ETHERTYPE_AT */

/*
 * Reconfiguring the network devices requires the net_config privilege
 * in Solaris 10+.
 */
extern int secpolicy_net_config(const cred_t *, boolean_t);

#include <sys/miiregs.h>                /* by fjlite out of intel       */

#include "bge.h"
#include "bge_hw.h"

/*
 * Compile-time feature switches ...
 */
#define BGE_DO_PPIO             0       /* peek/poke ioctls             */
#define BGE_RX_SOFTINT          0       /* softint per receive ring     */
#define BGE_CHOOSE_SEND_METHOD  0       /* send by copying only         */

/*
 * NOTES:
```

```
 128  *
 129  * #defines:
 130  *
 131  *      BGE_PCI_CONFIG_RNUMBER and BGE_PCI_OPREGS_RNUMBER are the
 132  *      register-set numbers to use for the config space registers
 133  *      and the operating registers respectively.  On an OBP-based
 134  *      machine, regset 0 refers to CONFIG space, and regset 1 will
 135  *      be the operating registers in MEMORY space.  If an expansion
 136  *      ROM is fitted, it may appear as a further register set.
 137  *
 138  *      BGE_DMA_MODE defines the mode (STREAMING/CONSISTENT) used
 139  *      for the data buffers.  The descriptors are always set up
 140  *      in CONSISTENT mode.
 141  *
 142  *      BGE_HEADROOM defines how much space we'll leave in allocated
 143  *      mblks before the first valid data byte.  This should be chosen
 144  *      to be 2 modulo 4, so that once the ethernet header (14 bytes)
 145  *      has been stripped off, the packet data will be 4-byte aligned.
 146  *      The remaining space can be used by upstream modules to prepend
 147  *      any headers required.
 148  */

 150 #define BGE_PCI_CONFIG_RNUMBER  0
 151 #define BGE_PCI_OPREGS_RNUMBER  1
 152 #define BGE_PCI_APEREGS_RNUMBER 2
 153 #define BGE_DMA_MODE            DDI_DMA_STREAMING
 154 #define BGE_HEADROOM            34

 156 /*
 157  *      BGE_HALFTICK is half the period of the cyclic callback (in
 158  *      nanoseconds), chosen so that 0.5s <= cyclic period <= 1s.
 159  *      Other time values are derived as odd multiples of this value
 160  *      so that there's little chance of ambiguity w.r.t. which tick
 161  *      a timeout expires on.
 162  *
 163  *      BGE_PHY_STABLE_TIME is the period for which the contents of the
 164  *      PHY's status register must remain unchanging before we accept
 165  *      that the link has come up.  [Sometimes the link comes up, only
 166  *      to go down again within a short time as the autonegotiation
 167  *      process cycles through various options before finding the best
 168  *      compatible mode.  We don't want to report repeated link up/down
 169  *      cycles, so we wait until we think it's stable.]
 170  *
 171  *      BGE_SERDES_STABLE_TIME is the analogous value for the SerDes
 172  *      interface.  It's much shorter, 'cos the SerDes doesn't show
 173  *      these effects as much as the copper PHY.
 174  *
 175  *      BGE_LINK_SETTLE_TIME is the period during which we regard link
 176  *      up/down cycles as an normal event after resetting/reprogramming
 177  *      the PHY.  During this time, link up/down messages are sent to
 178  *      the log only, not the console.  At any other time, link change
 179  *      events are regarded as unexpected and sent to both console & log.
 180  *
 181  *      These latter two values have no theoretical justification, but
 182  *      are derived from observations and heuristics - the values below
 183  *      just seem to work quite well.
 184  */

 186 #define BGE_HALFTICK            268435456LL             /* 2**28 ns!    */
 187 #define BGE_CYCLIC_PERIOD       (4*BGE_HALFTICK)        /*    ~1.0s     */
 188 #define BGE_CYCLIC_TIMEOUT      drv_sectohz(1)          /*    ~1.0s     */
 188 #define BGE_CYCLIC_TIMEOUT      (drv_usectohz(1000000)) /*    ~1.0s     */
 189 #define BGE_SERDES_STABLE_TIME  (3*BGE_HALFTICK)        /*    ~0.8s     */
 190 #define BGE_PHY_STABLE_TIME     (11*BGE_HALFTICK)       /*    ~3.0s     */
 191 #define BGE_LINK_SETTLE_TIME    (111*BGE_HALFTICK)      /*   ~30.0s     */
```



```
 193 /*
 194  * Indices used to identify the different buffer rings internally
 195  */
 196 #define BGE_STD_BUFF_RING       0
 197 #define BGE_JUMBO_BUFF_RING     1
 198 #define BGE_MINI_BUFF_RING      2

 200 /*
 201  * Current implementation limits
 202  */
 203 #define BGE_BUFF_RINGS_USED     2               /* std & jumbo ring     */
 204                                                 /* for now              */
 205 #define BGE_RECV_RINGS_USED     16              /* up to 16 rtn rings   */
 206                                                 /* for now              */
 207 #define BGE_SEND_RINGS_USED     4               /* up to 4 tx rings     */
 208                                                 /* for now              */
 209 #define BGE_HASH_TABLE_SIZE     128             /* may be 256 later     */

 211 /*
 212  * Ring/buffer size parameters
 213  *
 214  * All of the (up to) 16 TX rings & and the corresponding buffers are the
 215  * same size.
 216  *
 217  * Each of the (up to) 3 receive producer (aka buffer) rings is a different
 218  * size and has different sized buffers associated with it too.
 219  *
 220  * The (up to) 16 receive return rings have no buffers associated with them.
 221  * The number of slots per receive return ring must be 2048 if the mini
 222  * ring is enabled, otherwise it may be 1024.  See Broadcom document
 223  * 570X-PG102-R page 56.
 224  *
 225  * Note: only the 5700 supported external memory (and therefore the mini
 226  * ring); the 5702/3/4 don't.  This driver doesn't support the original
 227  * 5700, so we won't ever use the mini ring capability.
 228  */

 230 #define BGE_SEND_RINGS_DEFAULT          1
 231 #define BGE_RECV_RINGS_DEFAULT          1

 233 #define BGE_SEND_BUFF_SIZE_DEFAULT      1536
 234 #define BGE_SEND_BUFF_SIZE_JUMBO        9022
 235 #define BGE_SEND_SLOTS_USED     512

 237 #define BGE_STD_BUFF_SIZE       1536            /* 0x600                */
 238 #define BGE_STD_SLOTS_USED      512

 240 #define BGE_JUMBO_BUFF_SIZE     9022            /* 9k                   */
 241 #define BGE_JUMBO_SLOTS_USED    256

 243 #define BGE_MINI_BUFF_SIZE      128             /* 64? 256?             */
 244 #define BGE_MINI_SLOTS_USED     0               /* must be 0; see above */

 246 #define BGE_RECV_BUFF_SIZE      0
 247 #if     BGE_MINI_SLOTS_USED > 0
 248 #define BGE_RECV_SLOTS_USED     2048            /* required             */
 249 #else
 250 #define BGE_RECV_SLOTS_USED     1024            /* could be 2048 anyway */
 251 #endif

 253 #define BGE_SEND_BUF_NUM        512
 254 #define BGE_SEND_BUF_ARRAY      16
 255 #define BGE_SEND_BUF_ARRAY_JUMBO        3
 256 #define BGE_SEND_BUF_MAX        (BGE_SEND_BUF_NUM*BGE_SEND_BUF_ARRAY)

 258 /*
```

```
 259  * PCI type. PCI-Express or PCI/PCIX
 260  */
 261 #define BGE_PCI         0
 262 #define BGE_PCI_E       1
 263 #define BGE_PCI_X       2

 265 /*
 266  * Statistic type. There are two type of statistic:
 267  * statistic block and statistic registers
 268  */
 269 #define BGE_STAT_BLK    1
 270 #define BGE_STAT_REG    2

 272 /*
 273  * MTU.for all chipsets ,the default is 1500 ,and some chipsets
 274  * support 9k jumbo frames size
 275  */
 276 #define BGE_DEFAULT_MTU         1500
 277 #define BGE_MAXIMUM_MTU         9000

 279 /*
 280  * Pad the h/w defined status block (which can be up to 80 bytes long)
 281  * to a power-of-two boundary
 282  */
 283 #define BGE_STATUS_PADDING      (128 - sizeof (bge_status_t))

 285 /*
 286  * On platforms which support DVMA, we can simply allocate one big piece
 287  * of memory for all the Tx buffers and another for the Rx buffers, and
 288  * then carve them up as required.  It doesn't matter if they aren't just
 289  * one physically contiguous piece each, because both the CPU *and* the
 290  * I/O device can see them *as though they were*.
 291  *
 292  * However, if only physically-addressed DMA is possible, this doesn't
 293  * work; we can't expect to get enough contiguously-addressed memory for
 294  * all the buffers of each type, so in this case we request a number of
 295  * smaller pieces, each still large enough for several buffers but small
 296  * enough to fit within "an I/O page" (e.g. 64K).
 297  *
 298  * The #define below specifies how many pieces of memory are to be used;
 299  * 16 has been shown to work on an i86pc architecture but this could be
 300  * different on other non-DVMA platforms ...
 301  */
 302 #ifdef  _DMA_USES_VIRTADDR
 303 #define BGE_SPLIT               1               /* no split required    */
 304 #else
 305 #if ((BGE_BUFF_RINGS_USED > 1) || (BGE_SEND_RINGS_USED > 1) || \
 306         (BGE_RECV_RINGS_USED > 1))
 307 #define BGE_SPLIT               128             /* split 128 ways       */
 308 #else
 309 #define BGE_SPLIT               16              /* split 16 ways        */
 310 #endif
 311 #endif  /* _DMA_USES_VIRTADDR */

 313 #define BGE_RECV_RINGS_SPLIT    (BGE_RECV_RINGS_MAX + 1)

 315 /*
 316  * STREAMS parameters
 317  */
 318 #define BGE_IDNUM               0               /* zero seems to work   */
 319 #define BGE_LOWAT               (256)
 320 #define BGE_HIWAT               (256*1024)

 322 /*
 323  * Basic data types, for clarity in distinguishing 'numbers'
 324  * used for different purposes ...
```



```
 325  *
 326  * A <bge_regno_t> is a register 'address' (offset) in any one of
 327  * various address spaces (PCI config space, PCI memory-mapped I/O
 328  * register space, MII registers, etc).  None of these exceeds 64K,
 329  * so we could use a 16-bit representation but pointer-sized objects
 330  * are more "natural" in most architectures; they seem to be handled
 331  * more efficiently on SPARC and no worse on x86.
 332  *
 333  * BGE_REGNO_NONE represents the non-existent value in this space.
 334  */
 335 typedef uintptr_t bge_regno_t;                  /* register # (offset)  */
 336 #define BGE_REGNO_NONE          (~(uintptr_t)0u)

 338 /*
 339  * Describes one chunk of allocated DMA-able memory
 340  *
 341  * In some cases, this is a single chunk as allocated from the system;
 342  * but we also use this structure to represent slices carved off such
 343  * a chunk.  Even when we don't really need all the information, we
 344  * use this structure as a convenient way of correlating the various
 345  * ways of looking at a piece of memory (kernel VA, IO space DVMA,
 346  * handle+offset, etc).
 347  */
 348 typedef struct {
 349         ddi_acc_handle_t        acc_hdl;        /* handle for memory    */
 350         void                    *mem_va;        /* CPU VA of memory     */
 351         uint32_t                nslots;         /* number of slots      */
 352         uint32_t                size;           /* size per slot        */
 353         size_t                  alength;        /* allocated size       */
 354                                                 /* >= product of above  */

 356         ddi_dma_handle_t        dma_hdl;        /* DMA handle           */
 357         offset_t                offset;         /* relative to handle   */
 358         ddi_dma_cookie_t        cookie;         /* associated cookie    */
 359         uint32_t                ncookies;       /* must be 1            */
 360         uint32_t                token;          /* arbitrary identifier */
 361 } dma_area_t;                                   /* 0x50 (80) bytes      */
_____unchanged_portion_omitted_
```

****
39312 Wed Aug 19 07:24:55 2015
new/usr/src/uts/common/io/blkdev/blkdev.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
1434 static int
1435 bd_check_state(bd_t *bd, enum dkio_state *state)
1436 {
1437         clock_t         when;

1439         for (;;) {

1441                 bd_update_state(bd);

1443                 mutex_enter(&bd->d_statemutex);

1445                 if (bd->d_state != *state) {
1446                         *state = bd->d_state;
1447                         mutex_exit(&bd->d_statemutex);
1448                         break;
1449                 }

1451                 when = drv_sectohz(1);
1451                 when = drv_usectohz(1000000);
1452                 if (cv_reltimedwait_sig(&bd->d_statecv, &bd->d_statemutex,
1453                     when, TR_CLOCK_TICK) == 0) {
1454                         mutex_exit(&bd->d_statemutex);
1455                         return (EINTR);
1456                 }

1458                 mutex_exit(&bd->d_statemutex);
1459         }

1461         return (0);
1462 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   25588 Wed Aug 19 07:24:55 2015
new/usr/src/uts/common/io/bnxe/bnxe_rx.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 204 boolean_t BnxeWaitForPacketsFromClient(um_device_t * pUM,
 205                                        int           cliIdx)
 206 {
 207     int i, idx, cnt=0, tot=0;

 209     switch (cliIdx)
 210     {
 211     case LM_CLI_IDX_FCOE:

 213         for (i = 0; i < 5; i++)
 214         {
 215             if ((cnt = pUM->rxq[FCOE_CID(&pUM->lm_dev)].rxBufUpInStack) == 0)
 216             {
 217                 break;
 218             }

 220             /* twiddle our thumbs for one second */
 221             delay(drv_sectohz(1));
 221             delay(drv_usectohz(1000000));
 222         }

 224         if (cnt)
 225         {
 226             BnxeLogWarn(pUM, "%d packets still held by FCoE (chain %d)!",
 227                         cnt, FCOE_CID(&pUM->lm_dev));
 228             return B_FALSE;
 229         }

 231         break;

 233     case LM_CLI_IDX_NDIS:

 235         tot = 0;

 237         LM_FOREACH_RSS_IDX(&pUM->lm_dev, idx)
 238         {
 239             for (i = 0; i < 5; i++)
 240             {
 241                 if ((cnt = pUM->rxq[idx].rxBufUpInStack) == 0)
 242                 {
 243                     break;
 244                 }

 246                 /* twiddle our thumbs for one second */
 247                 delay(drv_sectohz(1));
 247                 delay(drv_usectohz(1000000));
 248             }

 250             tot += cnt;
 251         }

 253         if (tot)
 254         {
 255             BnxeLogWarn(pUM, "%d packets still held by the stack (chain %d)!",
 256                         tot, idx);
 257             return B_FALSE;
 258         }
```



```
 260         break;

 262     default:

 264         BnxeLogWarn(pUM, "ERROR: Invalid cliIdx for BnxeWaitForPacketsFromClient
 265         break;
 266     }

 268     return B_TRUE;
 269 }
______unchanged_portion_omitted_
```

****
94028 Wed Aug 19 07:24:55 2015
new/usr/src/uts/common/io/bridge.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
3325 /*
3326  * This function allocates the main data structures for the bridge driver and
3327  * connects us into devfs.
3328  */
3329 static void
3330 bridge_inst_init(void)
3331 {
3332         bridge_scan_interval = drv_sectohz(5);
3333         bridge_fwd_age = drv_sectohz(25);
3332         bridge_scan_interval = 5 * drv_usectohz(1000000);
3333         bridge_fwd_age = 25 * drv_usectohz(1000000);

3335         rw_init(&bmac_rwlock, NULL, RW_DRIVER, NULL);
3336         list_create(&bmac_list, sizeof (bridge_mac_t),
3337             offsetof(bridge_mac_t, bm_node));
3338         list_create(&inst_list, sizeof (bridge_inst_t),
3339             offsetof(bridge_inst_t, bi_node));
3340         cv_init(&inst_cv, NULL, CV_DRIVER, NULL);
3341         mutex_init(&inst_lock, NULL, MUTEX_DRIVER, NULL);
3342         cv_init(&stream_ref_cv, NULL, CV_DRIVER, NULL);
3343         mutex_init(&stream_ref_lock, NULL, MUTEX_DRIVER, NULL);

3345         mac_bridge_vectors(bridge_xmit_cb, bridge_recv_cb, bridge_ref_cb,
3346             bridge_ls_cb);
3347 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  168241 Wed Aug 19 07:24:56 2015
new/usr/src/uts/common/io/bscv.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3175 /*
3176  * function     - bscv_stop_event_daemon
3177  * description  - Attempt to stop the event daemon thread.
3178  * inputs       - LOM soft state structure pointer
3179  * outputs      - DDI_SUCCESS OR DDI_FAILURE
3180  */
3181 static int
3182 bscv_stop_event_daemon(bscv_soft_state_t *ssp)
3183 {
3184         int try;
3185         int res = DDI_SUCCESS;

3187         mutex_enter(&ssp->task_mu);

3189         /* Wait for task daemon to stop running. */
3190         for (try = 0;
3191             ((ssp->task_flags & TASK_ALIVE_FLG) && try < 10);
3192             try++) {
3193                 /* Signal that the task daemon should stop */
3194                 ssp->task_flags |= TASK_STOP_FLG;
3195                 cv_signal(&ssp->task_cv);
3196                 /* Release task daemon lock. */
3197                 mutex_exit(&ssp->task_mu);
3198                 /*
3199                  * TODO - when the driver is modified to support
3200                  * system suspend or if this routine gets called
3201                  * during panic we should use drv_usecwait() rather
3202                  * than delay in those circumstances.
3203                  */
3204                 delay(drv_sectohz(1));
3204                 delay(drv_usectohz(1000000));
3205                 mutex_enter(&ssp->task_mu);
3206         }

3208         if (ssp->task_flags & TASK_ALIVE_FLG) {
3209                 res = DDI_FAILURE;
3210         }
3211         mutex_exit(&ssp->task_mu);

3213         return (res);
3214 }

3216 /*
3217  * function     - bscv_pause_event_daemon
3218  * description  - Attempt to pause the event daemon thread.
3219  * inputs       - LOM soft state structure pointer
3220  * outputs      - DDI_SUCCESS OR DDI_FAILURE
3221  */
3222 static int
3223 bscv_pause_event_daemon(bscv_soft_state_t *ssp)
3224 {
3225         int try;

3227         if (!(ssp->progress & BSCV_THREAD)) {
3228                 /* Nothing to do */
3229                 return (BSCV_SUCCESS);
3230         }

3232         BSCV_TRACE(ssp, 'D', "bscv_pause_event_daemon",
```



```
3233             "Attempting to pause event daemon");

3235         mutex_enter(&ssp->task_mu);
3236         /* Signal that the task daemon should pause */
3237         ssp->task_flags |= TASK_PAUSE_FLG;

3239         /* Wait for task daemon to pause. */
3240         for (try = 0;
3241             (!(ssp->task_flags & TASK_SLEEPING_FLG) &&
3242             (ssp->task_flags & TASK_ALIVE_FLG) &&
3243             try < 10);
3244             try++) {
3245                 /* Paranoia */
3246                 ssp->task_flags |= TASK_PAUSE_FLG;
3247                 cv_signal(&ssp->task_cv);
3248                 /* Release task daemon lock. */
3249                 mutex_exit(&ssp->task_mu);
3250                 delay(drv_sectohz(1));
3250                 delay(drv_usectohz(1000000));
3251                 mutex_enter(&ssp->task_mu);
3252         }
3253         if ((ssp->task_flags & TASK_SLEEPING_FLG) ||
3254             !(ssp->task_flags & TASK_ALIVE_FLG)) {
3255                 mutex_exit(&ssp->task_mu);
3256                 BSCV_TRACE(ssp, 'D', "bscv_pause_event_daemon",
3257                     "Pause event daemon - success");
3258                 return (BSCV_SUCCESS);
3259         }
3260         mutex_exit(&ssp->task_mu);
3261         BSCV_TRACE(ssp, 'D', "bscv_pause_event_daemon",
3262             "Pause event daemon - failed");
3263         return (BSCV_FAILURE);
3264 }
______unchanged_portion_omitted_
```

```
**********************************************************
   82509 Wed Aug 19 07:24:56 2015
new/usr/src/uts/common/io/comstar/port/fct/discovery.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/sysmacros.h>
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/disp.h>
#include <sys/byteorder.h>
#include <sys/varargs.h>
#include <sys/atomic.h>
#include <sys/sdt.h>

#include <sys/stmf.h>
#include <sys/stmf_ioctl.h>
#include <sys/portif.h>
#include <sys/fct.h>
#include <sys/fctio.h>

#include "fct_impl.h"
#include "discovery.h"

disc_action_t fct_handle_local_port_event(fct_i_local_port_t *iport);
disc_action_t fct_walk_discovery_queue(fct_i_local_port_t *iport);
disc_action_t fct_process_els(fct_i_local_port_t *iport,
    fct_i_remote_port_t *irp);
fct_status_t fct_send_accrjt(fct_cmd_t *cmd, uint8_t accrjt,
    uint8_t reason, uint8_t expl);
disc_action_t fct_link_init_complete(fct_i_local_port_t *iport);
fct_status_t fct_complete_previous_li_cmd(fct_i_local_port_t *iport);
fct_status_t fct_sol_plogi(fct_i_local_port_t *iport, uint32_t id,
    fct_cmd_t **ret_ppcmd, int implicit);
fct_status_t fct_sol_ct(fct_i_local_port_t *iport, uint32_t id,
    fct_cmd_t **ret_ppcmd, uint16_t opcode);
fct_status_t fct_ns_scr(fct_i_local_port_t *iport, uint32_t id,
    fct_cmd_t **ret_ppcmd);
```



```
static disc_action_t fct_check_cmdlist(fct_i_local_port_t *iport);
static disc_action_t fct_check_solcmd_queue(fct_i_local_port_t *iport);
static void fct_rscn_verify(fct_i_local_port_t *iport,
    uint8_t *rscn_req_payload, uint32_t rscn_req_size);
void fct_gid_cb(fct_i_cmd_t *icmd);

char *fct_els_names[] = { 0, "LS_RJT", "ACC", "PLOGI", "FLOGI", "LOGO",
                                "ABTX", "RCS", "RES", "RSS", "RSI", "ESTS",
                                "ESTC", "ADVC", "RTV", "RLS",
        /* 0x10 */              "ECHO", "TEST", "RRQ", "REC", "SRR", 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x20 */              "PRLI", "PRLO", "SCN", "TPLS",
                                "TPRLO", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x30 */              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x40 */              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x50 */              "PDISC", "FDISC", "ADISC", "RNC", "FARP",
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x60 */              "FAN", "RSCN", "SCR", 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0,
        /* 0x70 */              "LINIT", "LPC", "LSTS", 0, 0, 0, 0, 0,
                                "RNID", "RLIR", "LIRR", 0, 0, 0, 0, 0
                };

extern uint32_t fct_rscn_options;

/*
 * NOTE: if anybody drops the iport_worker_lock then they should not return
 * DISC_ACTION_NO_WORK. Which also means, dont drop the lock if you have
 * nothing to do. Or else return DISC_ACTION_RESCAN or DISC_ACTION_DELAY_RESCAN.
 * But you cannot be infinitly returning those so have some logic to
 * determine that there is nothing to do without dropping the lock.
 */
void
fct_port_worker(void *arg)
{
        fct_local_port_t        *port = (fct_local_port_t *)arg;
        fct_i_local_port_t      *iport = (fct_i_local_port_t *)
            port->port_fct_private;
        disc_action_t           suggested_action;
        clock_t                 dl, short_delay, long_delay;
        int64_t                 tmp_delay;

        iport->iport_cmdcheck_clock = ddi_get_lbolt() +
            drv_sectohz(FCT_CMDLIST_CHECK_SECONDS);
            drv_usectohz(FCT_CMDLIST_CHECK_SECONDS * 1000000);
        short_delay = drv_usectohz(10000);
        long_delay = drv_sectohz(1);
        long_delay = drv_usectohz(1000000);

        stmf_trace(iport->iport_alias, "iport is %p", iport);
        /* Discovery loop */
        mutex_enter(&iport->iport_worker_lock);
        atomic_or_32(&iport->iport_flags, IPORT_WORKER_RUNNING);
        while ((iport->iport_flags & IPORT_TERMINATE_WORKER) == 0) {
                suggested_action = DISC_ACTION_NO_WORK;
                /*
                 * Local port events are of the highest prioriy
                 */
                if (iport->iport_event_head) {
                        suggested_action |= fct_handle_local_port_event(iport);
                }

                /*
                 * We could post solicited ELSes to discovery queue.
                 * solicited CT will be processed inside fct_check_solcmd_queue
                 */
```

```
126                 if (iport->iport_solcmd_queue) {
127                         suggested_action |= fct_check_solcmd_queue(iport);
128                 }
129
130                 /*
131                  * All solicited and unsolicited ELS will be handled here
132                  */
133                 if (iport->iport_rpwe_head) {
134                         suggested_action |= fct_walk_discovery_queue(iport);
135                 }
136
137                 /*
138                  * We only process it when there's no outstanding link init CMD
139                  */
140                 if ((iport->iport_link_state == PORT_STATE_LINK_INIT_START) &&
141                     !(iport->iport_li_state & (LI_STATE_FLAG_CMD_WAITING |
142                     LI_STATE_FLAG_NO_LI_YET))) {
143                         suggested_action |= fct_process_link_init(iport);
144                 }
145
146                 /*
147                  * We process cmd aborting in the end
148                  */
149                 if (iport->iport_abort_queue) {
150                         suggested_action |= fct_cmd_terminator(iport);
151                 }
152
153                 /*
154                  * Check cmd max/free
155                  */
156                 if (iport->iport_cmdcheck_clock <= ddi_get_lbolt()) {
157                         suggested_action |= fct_check_cmdlist(iport);
158                         iport->iport_cmdcheck_clock = ddi_get_lbolt() +
159                             drv_sectohz(FCT_CMDLIST_CHECK_SECONDS);
159                             drv_usectohz(FCT_CMDLIST_CHECK_SECONDS * 1000000);
160                         iport->iport_max_active_ncmds = 0;
161                 }
162
163                 if (iport->iport_offline_prstate != FCT_OPR_DONE) {
164                         suggested_action |= fct_handle_port_offline(iport);
165                 }
166
167                 if (suggested_action & DISC_ACTION_RESCAN) {
168                         continue;
169                 } else if (suggested_action & DISC_ACTION_DELAY_RESCAN) {
170                         /*
171                          * This is not very optimum as whoever returned
172                          * DISC_ACTION_DELAY_RESCAN must have dropped the lock
173                          * and more things might have queued up. But since
174                          * we are only doing small delays, it only delays
175                          * things by a few ms, which is okey.
176                          */
177                         if (suggested_action & DISC_ACTION_USE_SHORT_DELAY) {
178                                 dl = short_delay;
179                         } else {
180                                 dl = long_delay;
181                         }
182                         atomic_or_32(&iport->iport_flags,
183                             IPORT_WORKER_DOING_TIMEDWAIT);
184                         (void) cv_reltimedwait(&iport->iport_worker_cv,
185                             &iport->iport_worker_lock, dl, TR_CLOCK_TICK);
186                         atomic_and_32(&iport->iport_flags,
187                             ~IPORT_WORKER_DOING_TIMEDWAIT);
188                 } else {
189                         atomic_or_32(&iport->iport_flags,
190                             IPORT_WORKER_DOING_WAIT);
```



```
191                         tmp_delay = (int64_t)(iport->iport_cmdcheck_clock -
192                             ddi_get_lbolt());
193                         if (tmp_delay < 0) {
194                                 tmp_delay = (int64_t)short_delay;
195                         }
196                         (void) cv_reltimedwait(&iport->iport_worker_cv,
197                             &iport->iport_worker_lock, (clock_t)tmp_delay,
198                             TR_CLOCK_TICK);
199                         atomic_and_32(&iport->iport_flags,
200                             ~IPORT_WORKER_DOING_WAIT);
201                 }
202         }
203
204         atomic_and_32(&iport->iport_flags, ~IPORT_WORKER_RUNNING);
205         mutex_exit(&iport->iport_worker_lock);
206 }
_____unchanged_portion_omitted_

537 /*
538  * iport_li_state can only be changed here and local_event
539  */
540 disc_action_t
541 fct_process_link_init(fct_i_local_port_t *iport)
542 {
543         fct_cmd_t       *cmd      = NULL;
544         char            *pname    = NULL;
545         uint8_t          elsop    = 0;
546         uint16_t         ctop     = 0;
547         uint32_t         wkdid    = 0;
548         int              implicit = 0;
549         int             force_login = 0;
550         disc_action_t    ret      = DISC_ACTION_RESCAN;
551         fct_link_info_t *li = &iport->iport_link_info;
552         char            topo[24], speed[4];
553
554         ASSERT(MUTEX_HELD(&iport->iport_worker_lock));
555
556 check_state_again:
557         switch (iport->iport_li_state & LI_STATE_MASK) {
558         case LI_STATE_DO_FLOGI:
559                 /* Is FLOGI even needed or already done ? */
560                 if ((iport->iport_link_info.port_no_fct_flogi) ||
561                     (IPORT_FLOGI_DONE(iport))) {
562                         iport->iport_li_state++;
563                         goto check_state_again;
564                 }
565                 fct_do_flogi(iport);
566                 break;
567
568         case LI_STATE_FINI_TOPOLOGY:
569                 fct_li_to_txt(li, topo, speed);
570                 cmn_err(CE_NOTE, "%s LINK UP, portid %x, topology %s,"
571                     "speed %s", iport->iport_alias, li->portid,
572                     topo, speed);
573                 if (li->port_topology !=
574                     iport->iport_link_old_topology) {
575                         if (iport->iport_nrps) {
576                                 /*
577                                  * rehash it if change from fabric to
578                                  * none fabric, vice versa
579                                  */
580                                 if ((li->port_topology ^
581                                     iport->iport_link_old_topology) &
582                                     PORT_TOPOLOGY_FABRIC_BIT) {
583                                         mutex_exit(&iport->iport_worker_lock);
584                                         fct_rehash(iport);
```

```
585                                     mutex_enter(&iport->iport_worker_lock);
586                             }
587                     }
588                     iport->iport_link_old_topology = li->port_topology;
589             }
590             /* Skip next level if topo is not N2N */
591             if (li->port_topology != PORT_TOPOLOGY_PT_TO_PT) {
592                     iport->iport_li_state += 2;
593                     atomic_and_32(&iport->iport_flags,
594                         ~IPORT_ALLOW_UNSOL_FLOGI);
595             } else {
596                     iport->iport_li_state++;
597                     iport->iport_login_retry = 0;
598                     iport->iport_li_cmd_timeout = ddi_get_lbolt() +
599                         drv_sectohz(25);
599                         drv_usectohz(25 * 1000000);
600             }
601             goto check_state_again;

603     case LI_STATE_N2N_PLOGI:
604             ASSERT(IPORT_FLOGI_DONE(iport));
605             ASSERT(iport->iport_link_info.port_topology ==
606                 PORT_TOPOLOGY_PT_TO_PT);
607             if (iport->iport_li_state & LI_STATE_FLAG_CMD_RETCHECK) {
608                     iport->iport_li_state &= ~LI_STATE_FLAG_CMD_RETCHECK;
609                     if (iport->iport_li_comp_status != FCT_SUCCESS) {
610                             iport->iport_login_retry++;
611                             if (iport->iport_login_retry >= 3) {
612                                     stmf_trace(iport->iport_alias, "Failing"
613                                         " to PLOGI to remote port in N2N "
614                                         " ret=%llx, forcing link down",
615                                         iport->iport_li_comp_status);
616                                     mutex_exit(&iport->iport_worker_lock);
617                                     fct_handle_event(iport->iport_port,
618                                         FCT_EVENT_LINK_DOWN, 0, 0);
619                                     mutex_enter(&iport->iport_worker_lock);
620                             }
621                     }
622             }
623             /* Find out if we need to do PLOGI at all */
624             if (iport->iport_nrps_login) {
625                     iport->iport_li_state++;
626                     atomic_and_32(&iport->iport_flags,
627                         ~IPORT_ALLOW_UNSOL_FLOGI);
628                     goto check_state_again;
629             }
630             if ((ddi_get_lbolt() >= iport->iport_li_cmd_timeout) &&
631                 (!fct_lport_has_bigger_wwn(iport))) {
632                     /* Cant wait forever */
633                     stmf_trace(iport->iport_alias, "N2N: Remote port is "
634                         "not logging in, forcing from our side");
635                     force_login = 1;
636             } else {
637                     force_login = 0;
638             }
639             if (force_login || fct_lport_has_bigger_wwn(iport)) {
640                     elsop    = ELS_OP_PLOGI;
641                     wkdid    = 1;
642                     iport->iport_link_info.portid = 0xEF;
643                     implicit = 0;
644                     iport->iport_li_state |= LI_STATE_FLAG_CMD_RETCHECK;
645             } else {
646                     ret = DISC_ACTION_DELAY_RESCAN;
647             }
648             break;
```



```
650     case LI_STATE_DO_FCLOGIN:
651             if (iport->iport_li_state & LI_STATE_FLAG_CMD_RETCHECK) {
652                     iport->iport_li_state &= ~LI_STATE_FLAG_CMD_RETCHECK;
653                     if (iport->iport_li_comp_status != FCT_SUCCESS) {
654                             /*
655                              * Fabric controller login failed. Just skip all
656                              * the fabric controller related cmds.
657                              */
658                             iport->iport_li_state = LI_STATE_DO_SCR + 1;
659                     } else {
660                             /*
661                              * Good. Now lets go to next state
662                              */
663                             iport->iport_li_state++;
664                     }
665                     goto check_state_again;
666             }
667             if (!IPORT_IN_NS_TOPO(iport)) {
668                     iport->iport_li_state = LI_STATE_DO_SCR + 1;
669                     goto check_state_again;
670             }

672             elsop    = ELS_OP_PLOGI;
673             wkdid    = FS_FABRIC_CONTROLLER;
674             implicit = 1;

676             /*
677              * We want to come back in the same state and check its ret
678              * We can't modify the state here
679              */
680             iport->iport_li_state |= LI_STATE_FLAG_CMD_RETCHECK;
681             break;

683     case LI_STATE_DO_SCR:
684             elsop = ELS_OP_SCR;
685             wkdid = FS_FABRIC_CONTROLLER;

687             /*
688              * We dont care about success of this state. Just go to
689              * next state upon completion.
690              */
691             iport->iport_li_state++;
692             break;

694     case LI_STATE_DO_NSLOGIN:
695             if (iport->iport_li_state & LI_STATE_FLAG_CMD_RETCHECK) {
696                     iport->iport_li_state &= ~LI_STATE_FLAG_CMD_RETCHECK;
697                     if (iport->iport_li_comp_status != FCT_SUCCESS) {
698                             iport->iport_li_state = LI_STATE_DO_RSNN + 1;
699                     } else {
700                             iport->iport_li_state++;
701                     }
702                     goto check_state_again;
703             }

705             if (!IPORT_IN_NS_TOPO(iport)) {
706                     iport->iport_li_state = LI_STATE_DO_RSNN + 1;
707                     goto check_state_again;
708             }

710             elsop                   = ELS_OP_PLOGI;
711             wkdid                   = FS_NAME_SERVER;
712             iport->iport_li_state   |= LI_STATE_FLAG_CMD_RETCHECK;
713             break;

715             /*
```

```
716                  * CT state
717                  */
718         case LI_STATE_DO_RNN:
719                 ctop = NS_RNN_ID;
720                 iport->iport_li_state++;
721                 break;

723         case LI_STATE_DO_RCS:
724                 ctop = NS_RCS_ID;
725                 iport->iport_li_state++;
726                 break;

728         case LI_STATE_DO_RFT:
729                 ctop = NS_RFT_ID;
730                 iport->iport_li_state++;
731                 break;

733         case LI_STATE_DO_RSPN:
734                 /*
735                  * Check if we need skip the state
736                  */
737                 pname = iport->iport_port->port_sym_port_name !=
738                     NULL ? iport->iport_port->port_sym_port_name : NULL;
739                 if (pname == NULL) {
740                         pname = iport->iport_port->port_default_alias !=
741                             NULL ? iport->iport_port->port_default_alias : NULL;
742                         iport->iport_port->port_sym_port_name = pname;
743                 }

745                 if (pname == NULL) {
746                         iport->iport_li_state++;
747                         goto check_state_again;
748                 }

750                 ctop = NS_RSPN_ID;
751                 iport->iport_li_state++;
752                 break;

754         case LI_STATE_DO_RSNN:
755                 ctop = NS_RSNN_NN;
756                 iport->iport_li_state++;
757                 break;

759         case LI_STATE_MAX:
760                 mutex_exit(&iport->iport_worker_lock);

762                 fct_handle_event(iport->iport_port,
763                     FCT_I_EVENT_LINK_INIT_DONE, 0, 0);

765                 mutex_enter(&iport->iport_worker_lock);
766                 break;

768         default:
769                 ASSERT(0);
770         }

772         if (elsop != 0) {
773                 cmd = fct_create_solels(iport->iport_port, NULL, implicit,
774                     elsop, wkdid, fct_link_init_cb);
775         } else if (ctop != 0) {
776                 cmd = fct_create_solct(iport->iport_port, NULL, ctop,
777                     fct_link_init_cb);
778         }

780         if (cmd) {
781                 iport->iport_li_state |= LI_STATE_FLAG_CMD_WAITING;
```



```
782                 mutex_exit(&iport->iport_worker_lock);

784                 fct_post_to_solcmd_queue(iport->iport_port, cmd);

786                 mutex_enter(&iport->iport_worker_lock);
787         }

789         return (ret);
790 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   98668 Wed Aug 19 07:24:56 2015
new/usr/src/uts/common/io/comstar/port/fct/fct.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3230 /*
3231  * Called by worker thread. The aim is to terminate the command
3232  * using whatever means it takes.
3233  * Called with worker lock held.
3234  */
3235 disc_action_t
3236 fct_cmd_terminator(fct_i_local_port_t *iport)
3237 {
3238         char                    info[FCT_INFO_LEN];
3239         clock_t                 endtime;
3240         fct_i_cmd_t             **ppicmd;
3241         fct_i_cmd_t             *icmd;
3242         fct_cmd_t               *cmd;
3243         fct_local_port_t        *port = iport->iport_port;
3244         disc_action_t           ret = DISC_ACTION_NO_WORK;
3245         fct_status_t            abort_ret;
3246         int                     fca_done, fct_done, cmd_implicit = 0;
3247         int                     flags;
3248         unsigned long long      st;

3250         /* Lets Limit each run to 20ms max. */
3251         endtime = ddi_get_lbolt() + drv_usectohz(20000);

3253         /* Start from where we left off last time */
3254         if (iport->iport_ppicmd_term) {
3255                 ppicmd = iport->iport_ppicmd_term;
3256                 iport->iport_ppicmd_term = NULL;
3257         } else {
3258                 ppicmd = &iport->iport_abort_queue;
3259         }

3261         /*
3262          * Once a command gets on discovery queue, this is the only thread
3263          * which can access it. So no need for the lock here.
3264          */
3265         mutex_exit(&iport->iport_worker_lock);

3267         while ((icmd = *ppicmd) != NULL) {
3268                 cmd = icmd->icmd_cmd;

3270                 /* Always remember that cmd->cmd_rp can be NULL */
3271                 if ((icmd->icmd_flags & (ICMD_KNOWN_TO_FCA |
3272                     ICMD_FCA_ABORT_CALLED)) == ICMD_KNOWN_TO_FCA) {
3273                         atomic_or_32(&icmd->icmd_flags, ICMD_FCA_ABORT_CALLED);
3274                         if (CMD_HANDLE_VALID(cmd->cmd_handle))
3275                                 flags = 0;
3276                         else
3277                                 flags = FCT_IOF_FORCE_FCA_DONE;
3278                         abort_ret = port->port_abort_cmd(port, cmd, flags);
3279                         if ((abort_ret != FCT_SUCCESS) &&
3280                             (abort_ret != FCT_ABORT_SUCCESS) &&
3281                             (abort_ret != FCT_NOT_FOUND)) {
3282                                 if (flags & FCT_IOF_FORCE_FCA_DONE) {
3283                                         /*
3284                                          * XXX trigger port fatal,
3285                                          * Abort the termination, and shutdown
3286                                          * svc will trigger fct_cmd_termination
3287                                          * again.
3288                                          */
```



```
3289                                         (void) snprintf(info, sizeof (info),
3290                                             "fct_cmd_terminator:"
3291                                             " iport-%p, port_abort_cmd with "
3292                                             "FORCE_FCA_DONE failed",
3293                                             (void *)iport);
3294                                         (void) fct_port_shutdown(
3295                                             iport->iport_port,
3296                                             STMF_RFLAG_FATAL_ERROR |
3297                                             STMF_RFLAG_RESET, info);

3299                                         mutex_enter(&iport->iport_worker_lock);
3300                                         iport->iport_ppicmd_term = ppicmd;
3301                                         return (DISC_ACTION_DELAY_RESCAN);
3302                                 }
3303                                 atomic_and_32(&icmd->icmd_flags,
3304                                     ~ICMD_FCA_ABORT_CALLED);
3305                         } else if ((flags & FCT_IOF_FORCE_FCA_DONE) ||
3306                             (abort_ret == FCT_ABORT_SUCCESS) ||
3307                             (abort_ret == FCT_NOT_FOUND)) {
3308                                 atomic_and_32(&icmd->icmd_flags,
3309                                     ~ICMD_KNOWN_TO_FCA);
3310                         }
3311                         ret |= DISC_ACTION_DELAY_RESCAN;
3312                 } else if (icmd->icmd_flags & ICMD_IMPLICIT) {
3313                         if (cmd->cmd_type == FCT_CMD_SOL_ELS)
3314                                 cmd->cmd_comp_status = FCT_ABORTED;
3315                         atomic_or_32(&icmd->icmd_flags, ICMD_FCA_ABORT_CALLED);
3316                         cmd_implicit = 1;
3317                 }
3318                 if ((icmd->icmd_flags & ICMD_KNOWN_TO_FCA) == 0)
3319                         fca_done = 1;
3320                 else
3321                         fca_done = 0;
3322                 if ((icmd->icmd_flags & ICMD_IN_IRP_QUEUE) == 0)
3323                         fct_done = 1;
3324                 else
3325                         fct_done = 0;
3326                 if ((fca_done || cmd_implicit) && fct_done) {
3327                         mutex_enter(&iport->iport_worker_lock);
3328                         ASSERT(*ppicmd == icmd);
3329                         *ppicmd = (*ppicmd)->icmd_next;
3330                         mutex_exit(&iport->iport_worker_lock);
3331                         if ((cmd->cmd_type == FCT_CMD_RCVD_ELS) ||
3332                             (cmd->cmd_type == FCT_CMD_RCVD_ABTS)) {
3333                                 /* Free the cmd */
3334                                 fct_cmd_free(cmd);
3335                         } else if (cmd->cmd_type == FCT_CMD_SOL_ELS) {
3336                                 fct_handle_sol_els_completion(iport, icmd);
3337                                 if (icmd->icmd_flags & ICMD_IMPLICIT) {
3338                                         if (IS_LOGO_ELS(icmd)) {
3339                                                 /* IMPLICIT LOGO is special */
3340                                                 fct_cmd_free(cmd);
3341                                         }
3342                                 }
3343                         } else if (cmd->cmd_type == FCT_CMD_SOL_CT) {
3344                                 fct_sol_ct_t *ct = ICMD_TO_CT(icmd);

3346                                 /* Tell the caller that we are done */
3347                                 atomic_or_32(&icmd->icmd_flags,
3348                                     ICMD_CMD_COMPLETE);
3349                                 if (fct_netbuf_to_value(
3350                                     ct->ct_req_payload + 8, 2) == NS_GID_PN) {
3351                                         fct_i_remote_port_t *irp;

3353                                         rw_enter(&iport->iport_lock, RW_READER);
3354                                         irp = fct_lookup_irp_by_portwwn(iport,
```

```
3355                                        ct->ct_req_payload + 16);

3357                                    if (irp) {
3358                                            atomic_and_32(&irp->irp_flags,
3359                                                ~IRP_RSCN_QUEUED);
3360                                    }
3361                                    rw_exit(&iport->iport_lock);
3362                                }
3363                        } else {
3364                                ASSERT(0);
3365                        }
3366                } else {
3367                        clock_t timeout_ticks;
3368                        if (port->port_fca_abort_timeout)
3369                                timeout_ticks = drv_usectohz(
3370                                    port->port_fca_abort_timeout*1000);
3371                        else
3372                                /* 10 seconds by default */
3373                                timeout_ticks = drv_sectohz(10);
3373                                timeout_ticks = drv_usectohz(10 * 1000000);
3374                        if ((ddi_get_lbolt() >
3375                            (icmd->icmd_start_time+timeout_ticks)) &&
3376                            iport->iport_state == FCT_STATE_ONLINE) {
3377                                /* timeout, reset the port */
3378                                char cmd_type[10];
3379                                if (cmd->cmd_type == FCT_CMD_RCVD_ELS ||
3380                                    cmd->cmd_type == FCT_CMD_SOL_ELS) {
3381                                        fct_els_t *els = cmd->cmd_specific;
3382                                        (void) snprintf(cmd_type,
3383                                            sizeof (cmd_type), "%x.%x",
3384                                            cmd->cmd_type,
3385                                            els->els_req_payload[0]);
3386                                } else if (cmd->cmd_type == FCT_CMD_SOL_CT) {
3387                                        fct_sol_ct_t *ct = cmd->cmd_specific;
3388                                        (void) snprintf(cmd_type,
3389                                            sizeof (cmd_type), "%x.%02x%02x",
3390                                            cmd->cmd_type,
3391                                            ct->ct_req_payload[8],
3392                                            ct->ct_req_payload[9]);
3393                                } else {
3394                                        cmd_type[0] = 0;
3395                                }
3396                                st = cmd->cmd_comp_status;      /* gcc fix */
3397                                (void) snprintf(info, sizeof (info),
3398                                    "fct_cmd_terminator:"
3399                                    " iport-%p, cmd_type(0x%s),"
3400                                    " reason(%llx)", (void *)iport, cmd_type,
3401                                    st);
3402                                (void) fct_port_shutdown(port,
3403                                    STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET,
3404                                    info);
3405                        }
3406                        ppicmd = &((*ppicmd)->icmd_next);
3407                }

3409                if (ddi_get_lbolt() > endtime) {
3410                        mutex_enter(&iport->iport_worker_lock);
3411                        iport->iport_ppicmd_term = ppicmd;
3412                        return (DISC_ACTION_DELAY_RESCAN);
3413                }
3414        }
3415        mutex_enter(&iport->iport_worker_lock);
3416        if (iport->iport_abort_queue)
3417                return (DISC_ACTION_DELAY_RESCAN);
3418        if (ret == DISC_ACTION_NO_WORK)
3419                return (DISC_ACTION_RESCAN);
```



```
3420        return (ret);
3421 }
```

_____***unchanged_portion_omitted_***

****
94809 Wed Aug 19 07:24:56 2015
new/usr/src/uts/common/io/comstar/port/iscsit/iscsit.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
3309 /*
3310  * A separate thread is used to scan the staging queue on all the
3311  * sessions, If a delayed PDU does not arrive within a timeout, the
3312  * target will advance to the staged PDU that is next in sequence
3313  * and exceeded the threshold wait time. It is up to the initiator
3314  * to note that the target has not acknowledged a particular cmdsn
3315  * and take appropriate action.
3316  */
3317 /* ARGSUSED */
3318 static void
3319 iscsit_rxpdu_queue_monitor(void *arg)
3320 {
3321         iscsit_tgt_t    *tgt;
3322         iscsit_sess_t   *ist;

3324         mutex_enter(&iscsit_rxpdu_queue_monitor_mutex);
3325         iscsit_rxpdu_queue_monitor_thr_did = curthread->t_did;
3326         iscsit_rxpdu_queue_monitor_thr_running = B_TRUE;
3327         cv_signal(&iscsit_rxpdu_queue_monitor_cv);

3329         while (iscsit_rxpdu_queue_monitor_thr_running) {
3330                 ISCSIT_GLOBAL_LOCK(RW_READER);
3331                 for (tgt = avl_first(&iscsit_global.global_target_list);
3332                     tgt != NULL;
3333                     tgt = AVL_NEXT(&iscsit_global.global_target_list, tgt)) {
3334                         mutex_enter(&tgt->target_mutex);
3335                         for (ist = avl_first(&tgt->target_sess_list);
3336                             ist != NULL;
3337                             ist = AVL_NEXT(&tgt->target_sess_list, ist)) {

3339                                 iscsit_rxpdu_queue_monitor_session(ist);
3340                         }
3341                         mutex_exit(&tgt->target_mutex);
3342                 }
3343                 ISCSIT_GLOBAL_UNLOCK();
3344                 if (iscsit_rxpdu_queue_monitor_thr_running == B_FALSE) {
3345                         break;
3346                 }
3347                 (void) cv_reltimedwait(&iscsit_rxpdu_queue_monitor_cv,
3348                     &iscsit_rxpdu_queue_monitor_mutex,
3349                     drv_sectohz(ISCSIT_RXPDU_QUEUE_MONITOR_INTERVAL),
3349                     ISCSIT_RXPDU_QUEUE_MONITOR_INTERVAL * drv_usectohz(1000000),
3350                     TR_CLOCK_TICK);
3351         }
3352         mutex_exit(&iscsit_rxpdu_queue_monitor_mutex);
3353         thread_exit();
3354 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   92783 Wed Aug 19 07:24:57 2015
new/usr/src/uts/common/io/comstar/port/iscsit/iscsit_isns.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 490 int
 491 iscsit_isns_init(iscsit_hostinfo_t *hostinfo)
 492 {
 493         mutex_init(&iscsit_global.global_isns_cfg.isns_mutex, NULL,
 494             MUTEX_DEFAULT, NULL);

 496         ISNS_GLOBAL_LOCK();
 497         mutex_init(&iscsit_isns_mutex, NULL, MUTEX_DEFAULT, NULL);

 499         iscsit_global.global_isns_cfg.isns_state = B_FALSE;
 500         list_create(&iscsit_global.global_isns_cfg.isns_svrs,
 501             sizeof (iscsit_isns_svr_t), offsetof(iscsit_isns_svr_t, svr_ln));
 502         avl_create(&isns_tpg_portals, isnst_portal_avl_compare,
 503             sizeof (isns_portal_t), offsetof(isns_portal_t, portal_node));
 504         avl_create(&isns_all_portals, isnst_portal_avl_compare,
 505             sizeof (isns_portal_t), offsetof(isns_portal_t, portal_node));
 506         num_default_portals = 0;
 507         if (hostinfo->length > ISCSIT_MAX_HOSTNAME_LEN)
 508                 hostinfo->length = ISCSIT_MAX_HOSTNAME_LEN;
 509         isns_eid = kmem_alloc(hostinfo->length, KM_SLEEP);
 510         (void) strlcpy(isns_eid, hostinfo->fqhn, hostinfo->length);
 511         avl_create(&isns_target_list, isnst_tgt_avl_compare,
 512             sizeof (isns_target_t), offsetof(isns_target_t, target_node));

 514         /* initialize isns client */
 515         mutex_init(&isns_monitor_mutex, NULL, MUTEX_DEFAULT, NULL);
 516         mutex_init(&esi.esi_mutex, NULL, MUTEX_DEFAULT, NULL);
 517         isns_monitor_thr_id = NULL;
 518         monitor_idle_interval = drv_sectohz(ISNS_IDLE_TIME);
 518         monitor_idle_interval = ISNS_IDLE_TIME * drv_usectohz(1000000);
 519         cv_init(&isns_idle_cv, NULL, CV_DEFAULT, NULL);
 520         cv_init(&esi.esi_cv, NULL, CV_DEFAULT, NULL);
 521         xid = 0;
 522         ISNS_GLOBAL_UNLOCK();

 524         return (0);
 525 }
______unchanged_portion_omitted_

1062 /*
1063  * isnst_monitor -- the monitor thread for iSNS
1064  */
1065 /*ARGSUSED*/
1066 static void
1067 isnst_monitor(void *arg)
1068 {
1069         mutex_enter(&isns_monitor_mutex);
1070         isns_monitor_thr_did = curthread->t_did;
1071         isns_monitor_thr_running = B_TRUE;
1072         cv_signal(&isns_idle_cv);

1074         /*
1075          * Start with a short pause (5 sec) to allow all targets
1076          * to be registered before we send register-all.  This is
1077          * purely an optimization to cut down on the number of
1078          * messages we send to the iSNS server.
1079          */
1080         mutex_exit(&isns_monitor_mutex);
1081         delay(drv_sectohz(isns_initial_delay));
```



```
1081         delay(drv_usectohz(isns_initial_delay * 1000000));
1082         mutex_enter(&isns_monitor_mutex);

1084         /* Force an initialization of isns_all_portals */
1085         mutex_enter(&iscsit_isns_mutex);
1086         isns_portals_changed = B_TRUE;
1087         mutex_exit(&iscsit_isns_mutex);

1089         while (isns_monitor_thr_running) {

1091                 /* Update servers */
1092                 mutex_exit(&isns_monitor_mutex);
1093                 isnst_monitor_all_servers();
1094                 mutex_enter(&isns_monitor_mutex);

1096                 /* If something needs attention, go right to the top */
1097                 mutex_enter(&iscsit_isns_mutex);
1098                 if (isns_targets_changed || isns_portals_changed) {
1099                         DTRACE_PROBE(iscsit__isns__monitor__reenter);
1100                         mutex_exit(&iscsit_isns_mutex);
1101                         /* isns_monitor_mutex still held */
1102                         continue;
1103                 }
1104                 mutex_exit(&iscsit_isns_mutex);

1106                 /*
1107                  * Keep running until isns_monitor_thr_running is set to
1108                  * B_FALSE.
1109                  */
1110                 if (! isns_monitor_thr_running)
1111                         break;

1113                 DTRACE_PROBE(iscsit__isns__monitor__sleep);
1114                 (void) cv_reltimedwait(&isns_idle_cv, &isns_monitor_mutex,
1115                     monitor_idle_interval, TR_CLOCK_TICK);
1116                 DTRACE_PROBE1(iscsit__isns__monitor__wakeup,
1117                     boolean_t, isns_monitor_thr_running);
1118         }

1120         mutex_exit(&isns_monitor_mutex);

1122         /* Update the servers one last time for deregistration */
1123         isnst_monitor_all_servers();

1125         /* Clean up the all-portals list */
1126         ISNS_GLOBAL_LOCK();
1127         isnst_clear_default_portals();
1128         ISNS_GLOBAL_UNLOCK();

1130         /* terminate the thread at the last */
1131         thread_exit();
1132 }
______unchanged_portion_omitted_

2834 static size_t
2835 isnst_rcv_pdu(void *so, isns_pdu_t **pdu)
2836 {
2837         size_t          total_pdu_len;
2838         size_t          total_payload_len;
2839         size_t          payload_len;
2840         size_t          combined_len;
2841         isns_pdu_t      tmp_pdu_hdr;
2842         isns_pdu_t      *combined_pdu;
2843         uint8_t         *payload;
2844         uint8_t         *combined_payload;
2845         timeout_id_t    rcv_timer;
```

```
2846         uint16_t        flags;
2847         uint16_t        seq;

2849         ASSERT(! ISNS_GLOBAL_LOCK_HELD());

2851         *pdu = NULL;
2852         total_pdu_len = total_payload_len = 0;
2853         payload = NULL;
2854         seq = 0;

2856         do {
2857                 /* receive the pdu header */
2858                 rcv_timer = timeout(isnst_so_timeout, so,
2859                     drv_usectohz(isns_timeout_usec));
2860                 if (idm_sorecv(so, &tmp_pdu_hdr, ISNSP_HEADER_SIZE) != 0 ||
2861                     ntohs(tmp_pdu_hdr.seq) != seq) {
2862                         (void) untimeout(rcv_timer);
2863                         goto rcv_error;
2864                 }
2865                 (void) untimeout(rcv_timer);

2867                 /* receive the payload */
2868                 payload_len = ntohs(tmp_pdu_hdr.payload_len);
2869                 if (payload_len > ISNST_MAX_MSG_SIZE) {
2870                         goto rcv_error;
2871                 }
2872                 payload = kmem_alloc(payload_len, KM_NOSLEEP);
2873                 if (payload == NULL) {
2874                         goto rcv_error;
2875                 }
2876                 rcv_timer = timeout(isnst_so_timeout, so,
2877                     drv_sectohz(ISNS_RCV_TIMER_SECONDS));
2877                     drv_usectohz(ISNS_RCV_TIMER_SECONDS * 1000000));
2878                 if (idm_sorecv(so, payload, payload_len) != 0) {
2879                         (void) untimeout(rcv_timer);
2880                         goto rcv_error;
2881                 }
2882                 (void) untimeout(rcv_timer);

2884                 /* combine the pdu if it is not the first one */
2885                 if (total_pdu_len > 0) {
2886                         combined_len = total_pdu_len + payload_len;
2887                         combined_pdu = kmem_alloc(combined_len, KM_SLEEP);
2888                         if (combined_pdu == NULL) {
2889                                 goto rcv_error;
2890                         }
2891                         bcopy(*pdu, combined_pdu, total_pdu_len);
2892                         combined_payload =
2893                             &combined_pdu->payload[total_payload_len];
2894                         bcopy(payload, combined_payload, payload_len);
2895                         kmem_free(*pdu, total_pdu_len);
2896                         kmem_free(payload, payload_len);
2897                         *pdu = combined_pdu;
2898                         total_payload_len += payload_len;
2899                         total_pdu_len += payload_len;
2900                         (*pdu)->payload_len = htons(total_payload_len);
2901                 } else {
2902                         total_payload_len = payload_len;
2903                         total_pdu_len = ISNSP_HEADER_SIZE + payload_len;
2904                         *pdu = kmem_alloc(total_pdu_len, KM_NOSLEEP);
2905                         if (*pdu == NULL) {
2906                                 goto rcv_error;
2907                         }
2908                         bcopy(&tmp_pdu_hdr, *pdu, ISNSP_HEADER_SIZE);
2909                         bcopy(payload, &(*pdu)->payload[0], payload_len);
2910                         kmem_free(payload, payload_len);
```



```
2911                 }
2912                 payload = NULL;

2914                 /* the flags of pdu which is just received */
2915                 flags = ntohs(tmp_pdu_hdr.flags);

2917                 /* increase sequence number by one */
2918                 seq ++;
2919         } while ((flags & ISNS_FLAG_LAST_PDU) == 0);

2921         DTRACE_PROBE3(isnst__pdu__recv, uint16_t, ntohs((*pdu)->func_id),
2922             size_t, total_payload_len, caddr_t, *pdu);

2924         return (total_pdu_len);

2926 rcv_error:
2927         if (*pdu != NULL) {
2928                 kmem_free(*pdu, total_pdu_len);
2929                 *pdu = NULL;
2930         }
2931         if (payload != NULL) {
2932                 kmem_free(payload, payload_len);
2933         }
2934         return (0);
2935 }
_____unchanged_portion_omitted_

3047 /*
3048  * isnst_esi_thread
3049  *
3050  * This function listens on a socket for incoming connections from an
3051  * iSNS server until told to stop.
3052  */

3054 /*ARGSUSED*/
3055 static void
3056 isnst_esi_thread(void *arg)
3057 {
3058         ksocket_t               newso;
3059         struct sockaddr_in6     sin6;
3060         socklen_t               sin_addrlen;
3061         uint32_t                on = 1;
3062         int                     rc;
3063         isns_pdu_t              *pdu;
3064         size_t                  pl_size;

3066         bzero(&sin6, sizeof (struct sockaddr_in6));
3067         sin_addrlen = sizeof (struct sockaddr_in6);

3069         esi.esi_thread_did = curthread->t_did;

3071         mutex_enter(&esi.esi_mutex);

3073         /*
3074          * Mark the thread as running and the portal as no longer new.
3075          */
3076         esi.esi_thread_running = B_TRUE;
3077         cv_signal(&esi.esi_cv);

3079         while (esi.esi_enabled) {
3080                 /*
3081                  * Create a socket to listen for requests from the iSNS server.
3082                  */
3083                 if ((esi.esi_so = idm_socreate(PF_INET6, SOCK_STREAM, 0)) ==
3084                     NULL) {
3085                         ISNST_LOG(CE_WARN,
```

```
3086                            "isnst_esi_thread: Unable to create socket");
3087                        mutex_exit(&esi.esi_mutex);
3088                        delay(drv_sectohz(1));
3088                        delay(drv_usectohz(1000000));
3089                        mutex_enter(&esi.esi_mutex);
3090                        continue;
3091                }

3093                /*
3094                 * Set options, bind, and listen until we're told to stop
3095                 */
3096                bzero(&sin6, sizeof (sin6));
3097                sin6.sin6_family = AF_INET6;
3098                sin6.sin6_port = htons(0);
3099                sin6.sin6_addr = in6addr_any;

3101                (void) ksocket_setsockopt(esi.esi_so, SOL_SOCKET,
3102                    SO_REUSEADDR, (char *)&on, sizeof (on), CRED());

3104                if (ksocket_bind(esi.esi_so, (struct sockaddr *)&sin6,
3105                    sizeof (sin6), CRED()) != 0) {
3106                        ISNST_LOG(CE_WARN, "Unable to bind socket for ESI");
3107                        idm_sodestroy(esi.esi_so);
3108                        mutex_exit(&esi.esi_mutex);
3109                        delay(drv_sectohz(1));
3109                        delay(drv_usectohz(1000000));
3110                        mutex_enter(&esi.esi_mutex);
3111                        continue;
3112                }

3114                /*
3115                 * Get the port (sin6 is meaningless at this point)
3116                 */
3117                (void) ksocket_getsockname(esi.esi_so,
3118                    (struct sockaddr *)(&sin6), &sin_addrlen, CRED());
3119                esi.esi_port =
3120                    ntohs(((struct sockaddr_in6 *)(&sin6))->sin6_port);

3122                if ((rc = ksocket_listen(esi.esi_so, 5, CRED())) != 0) {
3123                        ISNST_LOG(CE_WARN, "isnst_esi_thread: listen "
3124                            "failure 0x%x", rc);
3125                        idm_sodestroy(esi.esi_so);
3126                        mutex_exit(&esi.esi_mutex);
3127                        delay(drv_sectohz(1));
3127                        delay(drv_usectohz(1000000));
3128                        mutex_enter(&esi.esi_mutex);
3129                        continue;
3130                }

3132                ksocket_hold(esi.esi_so);
3133                esi.esi_valid = B_TRUE;
3134                while (esi.esi_enabled) {
3135                        mutex_exit(&esi.esi_mutex);

3137                        DTRACE_PROBE3(iscsit__isns__esi__accept__wait,
3138                            boolean_t, esi.esi_enabled,
3139                            ksocket_t, esi.esi_so,
3140                            struct sockaddr_in6, &sin6);
3141                        if ((rc = ksocket_accept(esi.esi_so, NULL, NULL,
3142                            &newso, CRED())) != 0) {
3143                                mutex_enter(&esi.esi_mutex);
3144                                DTRACE_PROBE2(iscsit__isns__esi__accept__fail,
3145                                    int, rc, boolean_t, esi.esi_enabled);
3146                                /*
3147                                 * If we were interrupted with EINTR
3148                                 * it's not really a failure.
```



```
3149                                 */
3150                                ISNST_LOG(CE_WARN, "isnst_esi_thread: "
3151                                    "accept failure (0x%x)", rc);

3153                                if (rc == EINTR) {
3154                                        continue;
3155                                } else {
3156                                        break;
3157                                }
3158                        }
3159                        DTRACE_PROBE2(iscsit__isns__esi__accept,
3160                            boolean_t, esi.esi_enabled,
3161                            ksocket_t, newso);

3163                        pl_size = isnst_rcv_pdu(newso, &pdu);
3164                        if (pl_size == 0) {
3165                                ISNST_LOG(CE_WARN, "isnst_esi_thread: "
3166                                    "rcv_pdu failure");
3167                                idm_soshutdown(newso);
3168                                idm_sodestroy(newso);

3170                                mutex_enter(&esi.esi_mutex);
3171                                continue;
3172                        }

3174                        isnst_handle_esi_req(newso, pdu, pl_size);

3176                        idm_soshutdown(newso);
3177                        idm_sodestroy(newso);

3179                        mutex_enter(&esi.esi_mutex);
3180                }

3182                idm_soshutdown(esi.esi_so);
3183                ksocket_rele(esi.esi_so);
3184                esi.esi_valid = B_FALSE;

3186                /*
3187                 * If we're going to try to re-establish the listener then
3188                 * destroy this socket.  Otherwise isnst_esi_stop already
3189                 * destroyed it.
3190                 */
3191                if (esi.esi_enabled)
3192                        idm_sodestroy(esi.esi_so);
3193        }

3195        esi.esi_thread_running = B_FALSE;
3196        cv_signal(&esi.esi_cv);
3197        mutex_exit(&esi.esi_mutex);
3198 esi_thread_exit:
3199        thread_exit();
3200 }
______unchanged_portion_omitted_
```

****
52045 Wed Aug 19 07:24:57 2015
new/usr/src/uts/common/io/comstar/port/iscsit/iscsit_tgt.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
641 static void
642 iscsit_tgt_dereg_retry(void *arg)
643 {
644         iscsit_tgt_t *tgt = arg;

646         /*
647          * Rather than guaranteeing the target state machine code will not
648          * block for long periods of time (tying up this callout thread)
649          * we will queue a task on the taskq to send the retry event.
650          * If it fails we'll setup another timeout and try again later.
651          */
652         if (taskq_dispatch(iscsit_global.global_dispatch_taskq,
653             iscsit_tgt_dereg_task, tgt, DDI_NOSLEEP) == NULL) {
654                 /* Dispatch failed, try again later */
655                 (void) timeout(iscsit_tgt_dereg_retry, tgt,
656                     drv_sectohz(TGT_DEREG_RETRY_SECONDS));
656                     drv_usectohz(TGT_DEREG_RETRY_SECONDS * 1000000));
657         }
658 }
```

_____***unchanged_portion_omitted_***

****
22358 Wed Aug 19 07:24:57 2015
new/usr/src/uts/common/io/comstar/port/pppt/pppt_tgt.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 793 static void
 794 pppt_tgt_dereg_retry(void *arg)
 795 {
 796         pppt_tgt_t *tgt = arg;

 798         /*
 799          * Rather than guaranteeing the target state machine code will not
 800          * block for long periods of time (tying up this callout thread)
 801          * we will queue a task on the taskq to send the retry event.
 802          * If it fails we'll setup another timeout and try again later.
 803          */
 804         if (taskq_dispatch(pppt_global.global_dispatch_taskq,
 805             pppt_tgt_dereg_task, tgt, KM_NOSLEEP) == NULL) {
 806                 /* Dispatch failed, try again later */
 807                 (void) timeout(pppt_tgt_dereg_retry, tgt,
 808                     drv_sectohz(TGT_DEREG_RETRY_SECONDS));
 808                     drv_usectohz(TGT_DEREG_RETRY_SECONDS * 1000000));
 809         }
 810 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  180885 Wed Aug 19 07:24:57 2015
new/usr/src/uts/common/io/comstar/port/qlt/qlt.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


4956 /*
4957  * All QLT_FIRMWARE_* will mainly be handled in this function
4958  * It can not be called in interrupt context
4959  *
4960  * FWDUMP's purpose is to serve ioctl, so we will use qlt_ioctl_flags
4961  * and qlt_ioctl_lock
4962  */
4963 static fct_status_t
4964 qlt_firmware_dump(fct_local_port_t *port, stmf_state_change_info_t *ssci)
4965 {
4966         qlt_state_t     *qlt = (qlt_state_t *)port->port_fca_private;
4967         int             i;
4968         int             retries, n;
4969         uint_t          size_left;
4970         char            c = ' ';
4971         uint32_t        addr, endaddr, words_to_read;
4972         caddr_t         buf;
4973         fct_status_t    ret;

4975         mutex_enter(&qlt->qlt_ioctl_lock);
4976         /*
4977          * To make sure that there's no outstanding dumping task
4978          */
4979         if (qlt->qlt_ioctl_flags & QLT_FWDUMP_INPROGRESS) {
4980                 mutex_exit(&qlt->qlt_ioctl_lock);
4981                 EL(qlt, "qlt_ioctl_flags=%xh, inprogress\n",
4982                     qlt->qlt_ioctl_flags);
4983                 EL(qlt, "outstanding\n");
4984                 return (FCT_FAILURE);
4985         }

4987         /*
4988          * To make sure not to overwrite existing dump
4989          */
4990         if ((qlt->qlt_ioctl_flags & QLT_FWDUMP_ISVALID) &&
4991             !(qlt->qlt_ioctl_flags & QLT_FWDUMP_TRIGGERED_BY_USER) &&
4992             !(qlt->qlt_ioctl_flags & QLT_FWDUMP_FETCHED_BY_USER)) {
4993                 /*
4994                  * If we have alreay one dump, but it's not triggered by user
4995                  * and the user hasn't fetched it, we shouldn't dump again.
4996                  */
4997                 mutex_exit(&qlt->qlt_ioctl_lock);
4998                 EL(qlt, "qlt_ioctl_flags=%xh, already done\n",
4999                     qlt->qlt_ioctl_flags);
5000                 cmn_err(CE_NOTE, "qlt(%d): Skipping firmware dump as there "
5001                     "is one already outstanding.", qlt->instance);
5002                 return (FCT_FAILURE);
5003         }
5004         qlt->qlt_ioctl_flags |= QLT_FWDUMP_INPROGRESS;
5005         if (ssci->st_rflags & STMF_RFLAG_USER_REQUEST) {
5006                 qlt->qlt_ioctl_flags |= QLT_FWDUMP_TRIGGERED_BY_USER;
5007         } else {
5008                 qlt->qlt_ioctl_flags &= ~QLT_FWDUMP_TRIGGERED_BY_USER;
5009         }
5010         mutex_exit(&qlt->qlt_ioctl_lock);

5012         size_left = QLT_FWDUMP_BUFSIZE;
5013         if (!qlt->qlt_fwdump_buf) {
```



```
5014                 ASSERT(!(qlt->qlt_ioctl_flags & QLT_FWDUMP_ISVALID));
5015                 /*
5016                  * It's the only place that we allocate buf for dumping. After
5017                  * it's allocated, we will use it until the port is detached.
5018                  */
5019                 qlt->qlt_fwdump_buf = kmem_zalloc(size_left, KM_SLEEP);
5020         }

5022         /*
5023          * Start to dump firmware
5024          */
5025         buf = (caddr_t)qlt->qlt_fwdump_buf;

5027         /*
5028          * Print the ISP firmware revision number and attributes information
5029          * Read the RISC to Host Status register
5030          */
5031         n = (int)snprintf(buf, size_left, "ISP FW Version %d.%02d.%02d "
5032             "Attributes %04x\n\nR2H Status Register\n%08x",
5033             qlt->fw_major, qlt->fw_minor,
5034             qlt->fw_subminor, qlt->fw_attr, REG_RD32(qlt, REG_RISC_STATUS));
5035         buf += n; size_left -= n;

5037         /*
5038          * Before pausing the RISC, make sure no mailbox can execute
5039          */
5040         mutex_enter(&qlt->mbox_lock);
5041         if (qlt->mbox_io_state != MBOX_STATE_UNKNOWN) {
5042                 /*
5043                  * Wait to grab the mailboxes
5044                  */
5045                 for (retries = 0; (qlt->mbox_io_state != MBOX_STATE_READY) &&
5046                     (qlt->mbox_io_state != MBOX_STATE_UNKNOWN); retries++) {
5047                         (void) cv_timedwait(&qlt->mbox_cv, &qlt->mbox_lock,
5048                             ddi_get_lbolt() + drv_sectohz(1));
5048                             ddi_get_lbolt() + drv_usectohz(1000000));
5049                         if (retries > 5) {
5050                                 mutex_exit(&qlt->mbox_lock);
5051                                 EL(qlt, "can't drain out mailbox commands\n");
5052                                 goto dump_fail;
5053                         }
5054                 }
5055                 qlt->mbox_io_state = MBOX_STATE_UNKNOWN;
5056                 cv_broadcast(&qlt->mbox_cv);
5057         }
5058         mutex_exit(&qlt->mbox_lock);

5060         /*
5061          * Pause the RISC processor
5062          */
5063         REG_WR32(qlt, REG_HCCR, HCCR_CMD(SET_RISC_PAUSE));

5065         /*
5066          * Wait for the RISC processor to pause
5067          */
5068         for (i = 0; i < 200; i++) {
5069                 if (REG_RD32(qlt, REG_RISC_STATUS) & 0x100) {
5070                         break;
5071                 }
5072                 drv_usecwait(1000);
5073         }
5074         if (i == 200) {
5075                 EL(qlt, "can't pause\n");
5076                 return (FCT_FAILURE);
5077         }
```

```
5079         if ((!qlt->qlt_25xx_chip) && (!qlt->qlt_81xx_chip)) {
5080                 goto over_25xx_specific_dump;
5081         }
5082         n = (int)snprintf(buf, size_left, "\n\nHostRisc registers\n");
5083         buf += n; size_left -= n;
5084         REG_WR32(qlt, 0x54, 0x7000);
5085         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5086         buf += n; size_left -= n;
5087         REG_WR32(qlt, 0x54, 0x7010);
5088         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5089         buf += n; size_left -= n;
5090         REG_WR32(qlt, 0x54, 0x7C00);

5092         n = (int)snprintf(buf, size_left, "\nPCIe registers\n");
5093         buf += n; size_left -= n;
5094         REG_WR32(qlt, 0xC0, 0x1);
5095         n = qlt_fwdump_dump_regs(qlt, buf, 0xc4, 3, size_left);
5096         buf += n; size_left -= n;
5097         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 1, size_left);
5098         buf += n; size_left -= n;
5099         REG_WR32(qlt, 0xC0, 0x0);

5101 over_25xx_specific_dump:;
5102         n = (int)snprintf(buf, size_left, "\n\nHost Interface Registers\n");
5103         buf += n; size_left -= n;
5104         /*
5105          * Capture data from 32 regsiters
5106          */
5107         n = qlt_fwdump_dump_regs(qlt, buf, 0, 32, size_left);
5108         buf += n; size_left -= n;

5110         /*
5111          * Disable interrupts
5112          */
5113         REG_WR32(qlt, 0xc, 0);

5115         /*
5116          * Shadow registers
5117          */
5118         n = (int)snprintf(buf, size_left, "\nShadow Registers\n");
5119         buf += n; size_left -= n;

5121         REG_WR32(qlt, 0x54, 0xF70);
5122         addr = 0xb0000000;
5123         for (i = 0; i < 0xb; i++) {
5124                 if ((!qlt->qlt_25xx_chip) &&
5125                     (!qlt->qlt_81xx_chip) &&
5126                     (i >= 7)) {
5127                         break;
5128                 }
5129                 if (i && ((i & 7) == 0)) {
5130                         n = (int)snprintf(buf, size_left, "\n");
5131                         buf += n; size_left -= n;
5132                 }
5133                 REG_WR32(qlt, 0xF0, addr);
5134                 n = (int)snprintf(buf, size_left, "%08x ", REG_RD32(qlt, 0xFC));
5135                 buf += n; size_left -= n;
5136                 addr += 0x100000;
5137         }

5139         if ((qlt->qlt_25xx_chip) || (qlt->qlt_81xx_chip)) {
5140                 REG_WR32(qlt, 0x54, 0x10);
5141                 n = (int)snprintf(buf, size_left,
5142                     "\n\nRISC IO Register\n%08x", REG_RD32(qlt, 0xC0));
5143                 buf += n; size_left -= n;
5144         }
```



```
5146         /*
5147          * Mailbox registers
5148          */
5149         n = (int)snprintf(buf, size_left, "\n\nMailbox Registers\n");
5150         buf += n; size_left -= n;
5151         for (i = 0; i < 32; i += 2) {
5152                 if ((i + 2) & 15) {
5153                         c = ' ';
5154                 } else {
5155                         c = '\n';
5156                 }
5157                 n = (int)snprintf(buf, size_left, "%04x %04x%c",
5158                     REG_RD16(qlt, 0x80 + (i << 1)),
5159                     REG_RD16(qlt, 0x80 + ((i+1) << 1)), c);
5160                 buf += n; size_left -= n;
5161         }

5163         /*
5164          * Transfer sequence registers
5165          */
5166         n = (int)snprintf(buf, size_left, "\nXSEQ GP Registers\n");
5167         buf += n; size_left -= n;

5169         REG_WR32(qlt, 0x54, 0xBF00);
5170         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5171         buf += n; size_left -= n;
5172         REG_WR32(qlt, 0x54, 0xBF10);
5173         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5174         buf += n; size_left -= n;
5175         REG_WR32(qlt, 0x54, 0xBF20);
5176         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5177         buf += n; size_left -= n;
5178         REG_WR32(qlt, 0x54, 0xBF30);
5179         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5180         buf += n; size_left -= n;
5181         REG_WR32(qlt, 0x54, 0xBF40);
5182         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5183         buf += n; size_left -= n;
5184         REG_WR32(qlt, 0x54, 0xBF50);
5185         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5186         buf += n; size_left -= n;
5187         REG_WR32(qlt, 0x54, 0xBF60);
5188         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5189         buf += n; size_left -= n;
5190         REG_WR32(qlt, 0x54, 0xBF70);
5191         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5192         buf += n; size_left -= n;
5193         n = (int)snprintf(buf, size_left, "\nXSEQ-0 registers\n");
5194         buf += n; size_left -= n;
5195         if ((qlt->qlt_25xx_chip) || (qlt->qlt_81xx_chip)) {
5196                 REG_WR32(qlt, 0x54, 0xBFC0);
5197                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5198                 buf += n; size_left -= n;
5199                 REG_WR32(qlt, 0x54, 0xBFD0);
5200                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5201                 buf += n; size_left -= n;
5202         }
5203         REG_WR32(qlt, 0x54, 0xBFE0);
5204         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5205         buf += n; size_left -= n;
5206         n = (int)snprintf(buf, size_left, "\nXSEQ-1 registers\n");
5207         buf += n; size_left -= n;
5208         REG_WR32(qlt, 0x54, 0xBFF0);
5209         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5210         buf += n; size_left -= n;
```

```
5212         /*
5213          * Receive sequence registers
5214          */
5215         n = (int)snprintf(buf, size_left, "\nRSEQ GP Registers\n");
5216         buf += n; size_left -= n;
5217         REG_WR32(qlt, 0x54, 0xFF00);
5218         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5219         buf += n; size_left -= n;
5220         REG_WR32(qlt, 0x54, 0xFF10);
5221         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5222         buf += n; size_left -= n;
5223         REG_WR32(qlt, 0x54, 0xFF20);
5224         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5225         buf += n; size_left -= n;
5226         REG_WR32(qlt, 0x54, 0xFF30);
5227         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5228         buf += n; size_left -= n;
5229         REG_WR32(qlt, 0x54, 0xFF40);
5230         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5231         buf += n; size_left -= n;
5232         REG_WR32(qlt, 0x54, 0xFF50);
5233         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5234         buf += n; size_left -= n;
5235         REG_WR32(qlt, 0x54, 0xFF60);
5236         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5237         buf += n; size_left -= n;
5238         REG_WR32(qlt, 0x54, 0xFF70);
5239         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5240         buf += n; size_left -= n;
5241         n = (int)snprintf(buf, size_left, "\nRSEQ-0 registers\n");
5242         buf += n; size_left -= n;
5243         if ((qlt->qlt_25xx_chip) || (qlt->qlt_81xx_chip)) {
5244                 REG_WR32(qlt, 0x54, 0xFFC0);
5245                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5246                 buf += n; size_left -= n;
5247         }
5248         REG_WR32(qlt, 0x54, 0xFFD0);
5249         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5250         buf += n; size_left -= n;
5251         n = (int)snprintf(buf, size_left, "\nRSEQ-1 registers\n");
5252         buf += n; size_left -= n;
5253         REG_WR32(qlt, 0x54, 0xFFE0);
5254         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5255         buf += n; size_left -= n;
5256         n = (int)snprintf(buf, size_left, "\nRSEQ-2 registers\n");
5257         buf += n; size_left -= n;
5258         REG_WR32(qlt, 0x54, 0xFFF0);
5259         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5260         buf += n; size_left -= n;

5262         if ((!qlt->qlt_25xx_chip) && (!qlt->qlt_81xx_chip))
5263                 goto over_aseq_regs;

5265         /*
5266          * Auxiliary sequencer registers
5267          */
5268         n = (int)snprintf(buf, size_left, "\nASEQ GP Registers\n");
5269         buf += n; size_left -= n;
5270         REG_WR32(qlt, 0x54, 0xB000);
5271         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5272         buf += n; size_left -= n;
5273         REG_WR32(qlt, 0x54, 0xB010);
5274         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5275         buf += n; size_left -= n;
5276         REG_WR32(qlt, 0x54, 0xB020);
```



```
5277         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5278         buf += n; size_left -= n;
5279         REG_WR32(qlt, 0x54, 0xB030);
5280         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5281         buf += n; size_left -= n;
5282         REG_WR32(qlt, 0x54, 0xB040);
5283         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5284         buf += n; size_left -= n;
5285         REG_WR32(qlt, 0x54, 0xB050);
5286         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5287         buf += n; size_left -= n;
5288         REG_WR32(qlt, 0x54, 0xB060);
5289         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5290         buf += n; size_left -= n;
5291         REG_WR32(qlt, 0x54, 0xB070);
5292         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5293         buf += n; size_left -= n;
5294         n = (int)snprintf(buf, size_left, "\nASEQ-0 registers\n");
5295         buf += n; size_left -= n;
5296         REG_WR32(qlt, 0x54, 0xB0C0);
5297         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5298         buf += n; size_left -= n;
5299         REG_WR32(qlt, 0x54, 0xB0D0);
5300         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5301         buf += n; size_left -= n;
5302         n = (int)snprintf(buf, size_left, "\nASEQ-1 registers\n");
5303         buf += n; size_left -= n;
5304         REG_WR32(qlt, 0x54, 0xB0E0);
5305         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5306         buf += n; size_left -= n;
5307         n = (int)snprintf(buf, size_left, "\nASEQ-2 registers\n");
5308         buf += n; size_left -= n;
5309         REG_WR32(qlt, 0x54, 0xB0F0);
5310         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5311         buf += n; size_left -= n;

5313 over_aseq_regs:;

5315         /*
5316          * Command DMA registers
5317          */
5318         n = (int)snprintf(buf, size_left, "\nCommand DMA registers\n");
5319         buf += n; size_left -= n;
5320         REG_WR32(qlt, 0x54, 0x7100);
5321         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5322         buf += n; size_left -= n;

5324         /*
5325          * Queues
5326          */
5327         n = (int)snprintf(buf, size_left,
5328             "\nRequest0 Queue DMA Channel registers\n");
5329         buf += n; size_left -= n;
5330         REG_WR32(qlt, 0x54, 0x7200);
5331         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 8, size_left);
5332         buf += n; size_left -= n;
5333         n = qlt_fwdump_dump_regs(qlt, buf, 0xe4, 7, size_left);
5334         buf += n; size_left -= n;

5336         n = (int)snprintf(buf, size_left,
5337             "\n\nResponse0 Queue DMA Channel registers\n");
5338         buf += n; size_left -= n;
5339         REG_WR32(qlt, 0x54, 0x7300);
5340         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 8, size_left);
5341         buf += n; size_left -= n;
5342         n = qlt_fwdump_dump_regs(qlt, buf, 0xe4, 7, size_left);
```

```
5343         buf += n; size_left -= n;

5345         n = (int)snprintf(buf, size_left,
5346             "\n\nRequest1 Queue DMA Channel registers\n");
5347         buf += n; size_left -= n;
5348         REG_WR32(qlt, 0x54, 0x7400);
5349         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 8, size_left);
5350         buf += n; size_left -= n;
5351         n = qlt_fwdump_dump_regs(qlt, buf, 0xe4, 7, size_left);
5352         buf += n; size_left -= n;

5354         /*
5355          * Transmit DMA registers
5356          */
5357         n = (int)snprintf(buf, size_left, "\n\nXMT0 Data DMA registers\n");
5358         buf += n; size_left -= n;
5359         REG_WR32(qlt, 0x54, 0x7600);
5360         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5361         buf += n; size_left -= n;
5362         REG_WR32(qlt, 0x54, 0x7610);
5363         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5364         buf += n; size_left -= n;
5365         n = (int)snprintf(buf, size_left, "\nXMT1 Data DMA registers\n");
5366         buf += n; size_left -= n;
5367         REG_WR32(qlt, 0x54, 0x7620);
5368         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5369         buf += n; size_left -= n;
5370         REG_WR32(qlt, 0x54, 0x7630);
5371         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5372         buf += n; size_left -= n;
5373         n = (int)snprintf(buf, size_left, "\nXMT2 Data DMA registers\n");
5374         buf += n; size_left -= n;
5375         REG_WR32(qlt, 0x54, 0x7640);
5376         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5377         buf += n; size_left -= n;
5378         REG_WR32(qlt, 0x54, 0x7650);
5379         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5380         buf += n; size_left -= n;
5381         n = (int)snprintf(buf, size_left, "\nXMT3 Data DMA registers\n");
5382         buf += n; size_left -= n;
5383         REG_WR32(qlt, 0x54, 0x7660);
5384         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5385         buf += n; size_left -= n;
5386         REG_WR32(qlt, 0x54, 0x7670);
5387         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5388         buf += n; size_left -= n;
5389         n = (int)snprintf(buf, size_left, "\nXMT4 Data DMA registers\n");
5390         buf += n; size_left -= n;
5391         REG_WR32(qlt, 0x54, 0x7680);
5392         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5393         buf += n; size_left -= n;
5394         REG_WR32(qlt, 0x54, 0x7690);
5395         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5396         buf += n; size_left -= n;
5397         n = (int)snprintf(buf, size_left, "\nXMT Data DMA Common registers\n");
5398         buf += n; size_left -= n;
5399         REG_WR32(qlt, 0x54, 0x76A0);
5400         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5401         buf += n; size_left -= n;

5403         /*
5404          * Receive DMA registers
5405          */
5406         n = (int)snprintf(buf, size_left,
5407             "\nRCV Thread 0 Data DMA registers\n");
5408         buf += n; size_left -= n;
```



```
5409         REG_WR32(qlt, 0x54, 0x7700);
5410         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5411         buf += n; size_left -= n;
5412         REG_WR32(qlt, 0x54, 0x7710);
5413         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5414         buf += n; size_left -= n;
5415         n = (int)snprintf(buf, size_left,
5416             "\nRCV Thread 1 Data DMA registers\n");
5417         buf += n; size_left -= n;
5418         REG_WR32(qlt, 0x54, 0x7720);
5419         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5420         buf += n; size_left -= n;
5421         REG_WR32(qlt, 0x54, 0x7730);
5422         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5423         buf += n; size_left -= n;

5425         /*
5426          * RISC registers
5427          */
5428         n = (int)snprintf(buf, size_left, "\nRISC GP registers\n");
5429         buf += n; size_left -= n;
5430         REG_WR32(qlt, 0x54, 0x0F00);
5431         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5432         buf += n; size_left -= n;
5433         REG_WR32(qlt, 0x54, 0x0F10);
5434         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5435         buf += n; size_left -= n;
5436         REG_WR32(qlt, 0x54, 0x0F20);
5437         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5438         buf += n; size_left -= n;
5439         REG_WR32(qlt, 0x54, 0x0F30);
5440         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5441         buf += n; size_left -= n;
5442         REG_WR32(qlt, 0x54, 0x0F40);
5443         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5444         buf += n; size_left -= n;
5445         REG_WR32(qlt, 0x54, 0x0F50);
5446         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5447         buf += n; size_left -= n;
5448         REG_WR32(qlt, 0x54, 0x0F60);
5449         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5450         buf += n; size_left -= n;
5451         REG_WR32(qlt, 0x54, 0x0F70);
5452         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5453         buf += n; size_left -= n;

5455         /*
5456          * Local memory controller registers
5457          */
5458         n = (int)snprintf(buf, size_left, "\nLMC registers\n");
5459         buf += n; size_left -= n;
5460         REG_WR32(qlt, 0x54, 0x3000);
5461         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5462         buf += n; size_left -= n;
5463         REG_WR32(qlt, 0x54, 0x3010);
5464         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5465         buf += n; size_left -= n;
5466         REG_WR32(qlt, 0x54, 0x3020);
5467         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5468         buf += n; size_left -= n;
5469         REG_WR32(qlt, 0x54, 0x3030);
5470         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5471         buf += n; size_left -= n;
5472         REG_WR32(qlt, 0x54, 0x3040);
5473         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5474         buf += n; size_left -= n;
```

```
5475         REG_WR32(qlt, 0x54, 0x3050);
5476         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5477         buf += n; size_left -= n;
5478         REG_WR32(qlt, 0x54, 0x3060);
5479         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5480         buf += n; size_left -= n;

5482         if ((qlt->qlt_25xx_chip) || (qlt->qlt_81xx_chip)) {
5483                 REG_WR32(qlt, 0x54, 0x3070);
5484                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5485                 buf += n; size_left -= n;
5486         }

5488         /*
5489          * Fibre protocol module regsiters
5490          */
5491         n = (int)snprintf(buf, size_left, "\nFPM hardware registers\n");
5492         buf += n; size_left -= n;
5493         REG_WR32(qlt, 0x54, 0x4000);
5494         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5495         buf += n; size_left -= n;
5496         REG_WR32(qlt, 0x54, 0x4010);
5497         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5498         buf += n; size_left -= n;
5499         REG_WR32(qlt, 0x54, 0x4020);
5500         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5501         buf += n; size_left -= n;
5502         REG_WR32(qlt, 0x54, 0x4030);
5503         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5504         buf += n; size_left -= n;
5505         REG_WR32(qlt, 0x54, 0x4040);
5506         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5507         buf += n; size_left -= n;
5508         REG_WR32(qlt, 0x54, 0x4050);
5509         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5510         buf += n; size_left -= n;
5511         REG_WR32(qlt, 0x54, 0x4060);
5512         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5513         buf += n; size_left -= n;
5514         REG_WR32(qlt, 0x54, 0x4070);
5515         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5516         buf += n; size_left -= n;
5517         REG_WR32(qlt, 0x54, 0x4080);
5518         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5519         buf += n; size_left -= n;
5520         REG_WR32(qlt, 0x54, 0x4090);
5521         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5522         buf += n; size_left -= n;
5523         REG_WR32(qlt, 0x54, 0x40A0);
5524         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5525         buf += n; size_left -= n;
5526         REG_WR32(qlt, 0x54, 0x40B0);
5527         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5528         buf += n; size_left -= n;
5529         if (qlt->qlt_81xx_chip) {
5530                 REG_WR32(qlt, 0x54, 0x40C0);
5531                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5532                 buf += n; size_left -= n;
5533                 REG_WR32(qlt, 0x54, 0x40D0);
5534                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5535                 buf += n; size_left -= n;
5536         }

5538         /*
5539          * Fibre buffer registers
5540          */
```



```
5541         n = (int)snprintf(buf, size_left, "\nFB hardware registers\n");
5542         buf += n; size_left -= n;
5543         REG_WR32(qlt, 0x54, 0x6000);
5544         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5545         buf += n; size_left -= n;
5546         REG_WR32(qlt, 0x54, 0x6010);
5547         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5548         buf += n; size_left -= n;
5549         REG_WR32(qlt, 0x54, 0x6020);
5550         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5551         buf += n; size_left -= n;
5552         REG_WR32(qlt, 0x54, 0x6030);
5553         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5554         buf += n; size_left -= n;
5555         REG_WR32(qlt, 0x54, 0x6040);
5556         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5557         buf += n; size_left -= n;
5558         REG_WR32(qlt, 0x54, 0x6100);
5559         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5560         buf += n; size_left -= n;
5561         REG_WR32(qlt, 0x54, 0x6130);
5562         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5563         buf += n; size_left -= n;
5564         REG_WR32(qlt, 0x54, 0x6150);
5565         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5566         buf += n; size_left -= n;
5567         REG_WR32(qlt, 0x54, 0x6170);
5568         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5569         buf += n; size_left -= n;
5570         REG_WR32(qlt, 0x54, 0x6190);
5571         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5572         buf += n; size_left -= n;
5573         REG_WR32(qlt, 0x54, 0x61B0);
5574         n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5575         buf += n; size_left -= n;
5576         if (qlt->qlt_81xx_chip) {
5577                 REG_WR32(qlt, 0x54, 0x61C0);
5578                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5579                 buf += n; size_left -= n;
5580         }
5581         if ((qlt->qlt_25xx_chip) || (qlt->qlt_81xx_chip)) {
5582                 REG_WR32(qlt, 0x54, 0x6F00);
5583                 n = qlt_fwdump_dump_regs(qlt, buf, 0xc0, 16, size_left);
5584                 buf += n; size_left -= n;
5585         }

5587         qlt->intr_sneak_counter = 10;
5588         mutex_enter(&qlt->intr_lock);
5589         (void) qlt_reset_chip(qlt);
5590         drv_usecwait(20);
5591         qlt->intr_sneak_counter = 0;
5592         mutex_exit(&qlt->intr_lock);

5594         /*
5595          * Memory
5596          */
5597         n = (int)snprintf(buf, size_left, "\nCode RAM\n");
5598         buf += n; size_left -= n;

5600         addr = 0x20000;
5601         endaddr = 0x22000;
5602         words_to_read = 0;
5603         while (addr < endaddr) {
5604                 words_to_read = MBOX_DMA_MEM_SIZE >> 2;
5605                 if ((words_to_read + addr) > endaddr) {
5606                         words_to_read = endaddr - addr;
```

```
5607                }
5608                if ((ret = qlt_read_risc_ram(qlt, addr, words_to_read)) !=
5609                    QLT_SUCCESS) {
5610                        EL(qlt, "Error reading risc ram - CODE RAM status="
5611                            "%llxh\n", ret);
5612                        goto dump_fail;
5613                }
5614
5615                n = qlt_dump_risc_ram(qlt, addr, words_to_read, buf, size_left);
5616                buf += n; size_left -= n;
5617
5618                if (size_left < 100000) {
5619                        EL(qlt, "run out of space - CODE RAM size_left=%d\n",
5620                            size_left);
5621                        goto dump_ok;
5622                }
5623                addr += words_to_read;
5624        }
5625
5626        n = (int)snprintf(buf, size_left, "\nExternal Memory\n");
5627        buf += n; size_left -= n;
5628
5629        addr = 0x100000;
5630        endaddr = (((uint32_t)(qlt->fw_endaddrhi)) << 16) | qlt->fw_endaddrlo;
5631        endaddr++;
5632        if (endaddr & 7) {
5633                endaddr = (endaddr + 7) & 0xFFFFFFF8;
5634        }
5635
5636        words_to_read = 0;
5637        while (addr < endaddr) {
5638                words_to_read = MBOX_DMA_MEM_SIZE >> 2;
5639                if ((words_to_read + addr) > endaddr) {
5640                        words_to_read = endaddr - addr;
5641                }
5642                if ((ret = qlt_read_risc_ram(qlt, addr, words_to_read)) !=
5643                    QLT_SUCCESS) {
5644                        EL(qlt, "Error reading risc ram - EXT RAM status="
5645                            "%llxh\n", ret);
5646                        goto dump_fail;
5647                }
5648                n = qlt_dump_risc_ram(qlt, addr, words_to_read, buf, size_left);
5649                buf += n; size_left -= n;
5650                if (size_left < 100000) {
5651                        EL(qlt, "run out of space - EXT RAM\n");
5652                        goto dump_ok;
5653                }
5654                addr += words_to_read;
5655        }
5656
5657        /*
5658         * Label the end tag
5659         */
5660        n = (int)snprintf(buf, size_left, "[<==END] ISP Debug Dump\n");
5661        buf += n; size_left -= n;
5662
5663        /*
5664         * Queue dumping
5665         */
5666        n = (int)snprintf(buf, size_left, "\nRequest Queue\n");
5667        buf += n; size_left -= n;
5668        n = qlt_dump_queue(qlt, qlt->queue_mem_ptr + REQUEST_QUEUE_OFFSET,
5669            REQUEST_QUEUE_ENTRIES, buf, size_left);
5670        buf += n; size_left -= n;
5671
5672        n = (int)snprintf(buf, size_left, "\nPriority Queue\n");
```



```
5673        buf += n; size_left -= n;
5674        n = qlt_dump_queue(qlt, qlt->queue_mem_ptr + PRIORITY_QUEUE_OFFSET,
5675            PRIORITY_QUEUE_ENTRIES, buf, size_left);
5676        buf += n; size_left -= n;
5677
5678        n = (int)snprintf(buf, size_left, "\nResponse Queue\n");
5679        buf += n; size_left -= n;
5680        n = qlt_dump_queue(qlt, qlt->queue_mem_ptr + RESPONSE_QUEUE_OFFSET,
5681            RESPONSE_QUEUE_ENTRIES, buf, size_left);
5682        buf += n; size_left -= n;
5683
5684        n = (int)snprintf(buf, size_left, "\nATIO queue\n");
5685        buf += n; size_left -= n;
5686        n = qlt_dump_queue(qlt, qlt->queue_mem_ptr + ATIO_QUEUE_OFFSET,
5687            ATIO_QUEUE_ENTRIES, buf, size_left);
5688        buf += n; size_left -= n;
5689
5690        /*
5691         * Label dump reason
5692         */
5693        n = (int)snprintf(buf, size_left, "\nFirmware dump reason: %s-%s\n",
5694            qlt->qlt_port_alias, ssci->st_additional_info);
5695        buf += n; size_left -= n;
5696
5697 dump_ok:
5698        EL(qlt, "left-%d\n", size_left);
5699
5700        mutex_enter(&qlt->qlt_ioctl_lock);
5701        qlt->qlt_ioctl_flags &=
5702            ~(QLT_FWDUMP_INPROGRESS | QLT_FWDUMP_FETCHED_BY_USER);
5703        qlt->qlt_ioctl_flags |= QLT_FWDUMP_ISVALID;
5704        mutex_exit(&qlt->qlt_ioctl_lock);
5705        return (FCT_SUCCESS);
5706
5707 dump_fail:
5708        EL(qlt, "dump not done\n");
5709        mutex_enter(&qlt->qlt_ioctl_lock);
5710        qlt->qlt_ioctl_flags &= QLT_IOCTL_FLAG_MASK;
5711        mutex_exit(&qlt->qlt_ioctl_lock);
5712        return (FCT_FAILURE);
5713 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   46926 Wed Aug 19 07:24:58 2015
new/usr/src/uts/common/io/comstar/port/srpt/srpt_stp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 297 /*
 298  * srpt_stp_alloc_port() - Allocate SCSI Target Port
 299  */
 300 srpt_target_port_t *
 301 srpt_stp_alloc_port(srpt_ioc_t *ioc, ib_guid_t guid)
 302 {
 303         stmf_status_t           status;
 304         srpt_target_port_t      *tgt;
 305         stmf_local_port_t       *lport;
 306         uint64_t                temp;

 308         if (ioc == NULL) {
 309                 SRPT_DPRINTF_L1("stp_alloc_port, NULL I/O Controller");
 310                 return (NULL);
 311         }

 313         SRPT_DPRINTF_L3("stp_alloc_port, allocate STMF local port");
 314         lport = stmf_alloc(STMF_STRUCT_STMF_LOCAL_PORT, sizeof (*tgt), 0);
 315         if (lport == NULL) {
 316                 SRPT_DPRINTF_L1("tgt_alloc_port, stmf_alloc failed");
 317                 return (NULL);
 318         }

 320         tgt = lport->lport_port_private;
 321         ASSERT(tgt != NULL);

 323         mutex_init(&tgt->tp_lock, NULL, MUTEX_DRIVER, NULL);

 325         mutex_init(&tgt->tp_ch_list_lock, NULL, MUTEX_DRIVER, NULL);
 326         cv_init(&tgt->tp_offline_complete, NULL, CV_DRIVER, NULL);
 327         list_create(&tgt->tp_ch_list, sizeof (srpt_channel_t),
 328             offsetof(srpt_channel_t, ch_stp_node));

 330         mutex_init(&tgt->tp_sess_list_lock, NULL, MUTEX_DRIVER, NULL);
 331         cv_init(&tgt->tp_sess_complete, NULL, CV_DRIVER, NULL);
 332         list_create(&tgt->tp_sess_list, sizeof (srpt_session_t),
 333             offsetof(srpt_session_t, ss_node));

 335         tgt->tp_state    = SRPT_TGT_STATE_OFFLINE;
 336         tgt->tp_drv_disabled  = 0;
 337         tgt->tp_srp_enabled   = 0;
 338         tgt->tp_lport    = lport;
 339         tgt->tp_ioc        = ioc;
 340         tgt->tp_ibt_svc_id = guid;
 341         tgt->tp_ibt_svc_desc.sd_handler = srpt_cm_hdlr;
 342         tgt->tp_ibt_svc_desc.sd_flags   = IBT_SRV_NO_FLAGS;
 343         temp = h2b64(tgt->tp_ibt_svc_id);
 344         bcopy(&temp, &tgt->tp_srp_port_id[0], 8);
 345         temp = h2b64(tgt->tp_ioc->ioc_guid);
 346         bcopy(&temp, &tgt->tp_srp_port_id[8], 8);

 348         tgt->tp_nports  = ioc->ioc_attr.hca_nports;
 349         tgt->tp_hw_port =
 350             kmem_zalloc(sizeof (srpt_hw_port_t) * tgt->tp_nports, KM_SLEEP);
 351         tgt->tp_num_active_ports = 0;
 352         tgt->tp_requested_state = SRPT_TGT_STATE_OFFLINE;

 354         tgt->tp_scsi_devid = srpt_stp_alloc_scsi_devid_desc(tgt->tp_ibt_svc_id);
```



```
 356         lport->lport_id = tgt->tp_scsi_devid;
 357         lport->lport_pp = srpt_ctxt->sc_pp;
 358         lport->lport_ds = ioc->ioc_stmf_ds;
 359         lport->lport_xfer_data  = &srpt_stp_xfer_data;
 360         lport->lport_send_status = &srpt_stp_send_status;
 361         lport->lport_task_free  = &srpt_stp_task_free;
 362         lport->lport_abort      = &srpt_stp_abort;
 363         lport->lport_abort_timeout = 300;       /* 5 minutes */
 364         lport->lport_task_poll  = &srpt_stp_task_poll;
 365         lport->lport_ctl        = &srpt_stp_ctl;
 366         lport->lport_info       = &srpt_stp_info;
 367         lport->lport_event_handler = &srpt_stp_event_handler;

 369         /* set up as alua participating port */
 370         stmf_set_port_alua(lport);

 372         SRPT_DPRINTF_L3("stp_alloc_port, register STMF LPORT");

 374 retry_registration:
 375         status = stmf_register_local_port(lport);
 376         if (status == STMF_SUCCESS) {
 377                 SRPT_DPRINTF_L3("stp_alloc_port, LPORT successfully"
 378                     " registered");
 379                 return (tgt);
 380         }

 382         if (status == STMF_BUSY) {
 383                 /*
 384                  * This is only done on an administrative thread of
 385                  * execution so it is ok to take a while.
 386                  */
 387                 SRPT_DPRINTF_L3("stp_alloc_port, delaying");
 388                 delay(drv_sectohz(2));
 388                 delay(2 * drv_usectohz(1000000));
 389                 goto retry_registration;
 390         }
 391         SRPT_DPRINTF_L1("stp_alloc_port, STMF register local port err(0x%llx)",
 392             (u_longlong_t)status);

 394         SRPT_DPRINTF_L3("stp_alloc_port, free STMF local port");
 395         cv_destroy(&tgt->tp_offline_complete);
 396         mutex_destroy(&tgt->tp_ch_list_lock);
 397         mutex_destroy(&tgt->tp_lock);
 398         if (tgt->tp_hw_port) {
 399                 kmem_free(tgt->tp_hw_port,
 400                     sizeof (srpt_hw_port_t) * tgt->tp_nports);
 401         }
 402         if (tgt->tp_scsi_devid) {
 403                 srpt_stp_free_scsi_devid_desc(tgt->tp_scsi_devid);
 404         }

 406         stmf_free(lport);

 408         return (NULL);
 409 }
______unchanged_portion_omitted_

 447 /*
 448  * srpt_stp_destroy_port()
 449  */
 450 stmf_status_t
 451 srpt_stp_destroy_port(srpt_target_port_t *tgt)
 452 {
 453         stmf_status_t           status;
 454         stmf_change_status_t    cstatus;
 455         uint64_t                guid;
```

```
457         ASSERT(tgt != NULL);
458         ASSERT(tgt->tp_lport != NULL);

460         SRPT_DPRINTF_L3("stp_destroy_port, de-register STMF LPORT");

462         mutex_enter(&tgt->tp_lock);
463         if (tgt->tp_drv_disabled != 0) {
464                 /* already being destroyed, get out now - should not happen */
465                 mutex_exit(&tgt->tp_lock);
466                 return (STMF_ALREADY);
467         }

469         tgt->tp_drv_disabled = 1;
470         guid = tgt->tp_ioc->ioc_guid;
471         mutex_exit(&tgt->tp_lock);

473         SRPT_DPRINTF_L2("stp_destroy_port: unbind and de-register"
474             " services for GUID(%016llx)", (u_longlong_t)guid);

476         cstatus.st_completion_status = STMF_SUCCESS;
477         cstatus.st_additional_info = NULL;

479         status = stmf_ctl(STMF_CMD_LPORT_OFFLINE, tgt->tp_lport, &cstatus);

481         /*
482          * Wait for asynchronous target off-line operation
483          * to complete and then deregister the target
484          * port.
485          */
486         mutex_enter(&tgt->tp_lock);
487         while (tgt->tp_state != SRPT_TGT_STATE_OFFLINE) {
488                 cv_wait(&tgt->tp_offline_complete, &tgt->tp_lock);
489         }
490         mutex_exit(&tgt->tp_lock);

492         SRPT_DPRINTF_L3("stp_destroy_port: IOC (0x%016llx) Target"
493             " SRP off-line complete", (u_longlong_t)guid);

495         /* loop waiting for all I/O to drain */
496         for (;;) {
497                 status = stmf_deregister_local_port(tgt->tp_lport);
498                 if (status == STMF_BUSY) {
499                         delay(drv_sectohz(1));
499                         delay(drv_usectohz(1000000));
500                 } else {
501                         break;
502                 }
503         }

505         if (status == STMF_SUCCESS) {
506                 SRPT_DPRINTF_L3("stp_destroy_port, LPORT de-register"
507                     " complete");
508         } else {
509                 /*
510                  * Something other than a BUSY error, this should not happen.
511                  */
512                 SRPT_DPRINTF_L1(
513                     "stp_destroy_port, de-register STMF error(0x%llx)",
514                     (u_longlong_t)status);
515         }

517         return (status);
518 }
```

*unchanged_portion_omitted_*

****
215708 Wed Aug 19 07:24:58 2015
new/usr/src/uts/common/io/comstar/stmf/stmf.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
4265 void
4266 stmf_do_ilu_timeouts(stmf_i_lu_t *ilu)
4267 {
4268         clock_t l = ddi_get_lbolt();
4269         clock_t ps = drv_sectohz(1);
4269         clock_t ps = drv_usectohz(1000000);
4270         stmf_i_scsi_task_t *itask;
4271         scsi_task_t *task;
4272         uint32_t to;

4274         mutex_enter(&ilu->ilu_task_lock);
4275         for (itask = ilu->ilu_tasks; itask != NULL;
4276             itask = itask->itask_lu_next) {
4277                 if (itask->itask_flags & (ITASK_IN_FREE_LIST |
4278                     ITASK_BEING_ABORTED)) {
4279                         continue;
4280                 }
4281                 task = itask->itask_task;
4282                 if (task->task_timeout == 0)
4283                         to = stmf_default_task_timeout;
4284                 else
4285                         to = task->task_timeout;
4286                 if ((itask->itask_start_time + (to * ps)) > l)
4287                         continue;
4288                 stmf_abort(STMF_QUEUE_TASK_ABORT, task,
4289                     STMF_TIMEOUT, NULL);
4290         }
4291         mutex_exit(&ilu->ilu_task_lock);
4292 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   11142 Wed Aug 19 07:24:58 2015
new/usr/src/uts/common/io/comstar/stmf/stmf_impl.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 157 #define STMF_AVG_ONLINE_INTERVAL        drv_sectohz(30)
 157 #define STMF_AVG_ONLINE_INTERVAL        (30 * drv_usectohz(1000000))

 159 #define MAX_IRPORT                      0x10000

 161 typedef struct stmf_i_remote_port {
 162         struct scsi_devid_desc  *irport_id;
 163         kmutex_t                irport_mutex;
 164         int                     irport_refcnt;
 165         id_t                    irport_instance;
 166         avl_node_t              irport_ln;
 167 } stmf_i_remote_port_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   26591 Wed Aug 19 07:24:58 2015
new/usr/src/uts/common/io/cpqary3/cpqary3_talk2ctlr.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 882 /*
 883  * Function     :       cpqary3_check_ctlr_init
 884  * Description  :       This routine checks to see if the HBA is initialized.
 885  * Called By    :       cpqary3_init_ctlr()
 886  * Parameters   :       per-controller
 887  * Calls        :       None
 888  * Return Values:       SUCCESS / FAILURE
 889  */
 890 uint8_t
 891 cpqary3_check_ctlr_init(cpqary3_t *cpqary3p)
 892 {
 893         int8_t                          retvalue;
 894         uint16_t                        i;
 895         uint32_t                        *ctlr_init;
 896         ddi_acc_handle_t                ctlr_init_handle;
 897         extern ddi_device_acc_attr_t    cpqary3_dev_attributes;

 899         RETURN_FAILURE_IF_NULL(cpqary3p);

 901         /*
 902          * Set up the mapping for a Register at offset 0xB0 from I2O Bar
 903          * The value 0xB0 taken from the CONFIGM utility.
 904          * It should read 0xffff0000 if the controller is initialized.
 905          * if not yet initialized, read it every second for 300 secs.
 906          * If not set even after 300 secs, return FAILURE.
 907          * If set, free the mapping and continue
 908          */
 909         retvalue = ddi_regs_map_setup(cpqary3p->dip, INDEX_PCI_BASE0,
 910             (caddr_t *)&ctlr_init, (offset_t)I2O_CTLR_INIT, 4,
 911             &cpqary3_dev_attributes, &ctlr_init_handle);

 913         if (retvalue != DDI_SUCCESS) {
 914                 if (DDI_REGS_ACC_CONFLICT == retvalue)
 915                         cmn_err(CE_WARN,
 916                             "CPQary3 : HBA Init Register Mapping Conflict");
 917                 cmn_err(CE_WARN,
 918                     "CPQary3 : HBA Init Regsiter Mapping Failed");
 919                 return (CPQARY3_FAILURE);
 920         }

 922         for (i = 0; i < 300; i++) {     /* loop for 300 seconds */
 923                 if (CISS_CTLR_INIT == ddi_get32(ctlr_init_handle, ctlr_init)) {
 924                         DTRACE_PROBE(ctlr_init_check_ready);
 925                         ddi_regs_map_free(&ctlr_init_handle);
 926                         break;
 927                 } else {
 928                         DTRACE_PROBE(ctlr_init_check_notready);
 929                         delay(drv_sectohz(1));
 929                         delay(drv_usectohz(1000000));
 930                 }
 931         }

 933         if (i >= 300) { /* HBA not initialized even after 300 seconds !!! */
 934                 ddi_regs_map_free(&ctlr_init_handle);
 935                 cmn_err(CE_WARN, "CPQary3 : %s NOT initialized !!! HBA may not "
 936                     "function properly. Please replace the hardware or check "
 937                     "the connections", cpqary3p->hba_name);
 938                 return (CPQARY3_FAILURE);
 939         }
```



```
 941         return (CPQARY3_SUCCESS);
 942 }
______unchanged_portion_omitted_
```

****
171726 Wed Aug 19 07:24:58 2015
new/usr/src/uts/common/io/e1000g/e1000g_main.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
4321 static void
4322 e1000g_timer_tx_resched(struct e1000g *Adapter)
4323 {
4324         e1000g_tx_ring_t *tx_ring = Adapter->tx_ring;

4326         rw_enter(&Adapter->chip_lock, RW_READER);

4328         if (tx_ring->resched_needed &&
4329             ((ddi_get_lbolt() - tx_ring->resched_timestamp) >
4330             drv_sectohz(1)) &&
4330             drv_usectohz(1000000)) &&
4331             (Adapter->e1000g_state & E1000G_STARTED) &&
4332             (tx_ring->tbd_avail >= DEFAULT_TX_NO_RESOURCE)) {
4333                 tx_ring->resched_needed = B_FALSE;
4334                 mac_tx_update(Adapter->mh);
4335                 E1000G_STAT(tx_ring->stat_reschedule);
4336                 E1000G_STAT(tx_ring->stat_timer_reschedule);
4337         }

4339         rw_exit(&Adapter->chip_lock);
4340 }
```

_____***unchanged_portion_omitted_***

```
4638 static void
4639 arm_watchdog_timer(struct e1000g *Adapter)
4640 {
4641         Adapter->watchdog_tid =
4642             timeout(e1000g_local_timer,
4643             (void *)Adapter, drv_sectohz(1));
4643             (void *)Adapter, 1 * drv_usectohz(1000000));
4644 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  157437 Wed Aug 19 07:24:59 2015
new/usr/src/uts/common/io/ecpp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 751 static int
 752 ecpp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
 753 {
 754         int             instance;
 755         struct ecppunit *pp;

 757         instance = ddi_get_instance(dip);

 759         switch (cmd) {
 760         case DDI_DETACH:
 761                 break;

 763         case DDI_SUSPEND:
 764                 if (!(pp = ddi_get_soft_state(ecppsoft_statep, instance))) {
 765                         return (DDI_FAILURE);
 766                 }

 768                 mutex_enter(&pp->umutex);
 769                 ASSERT(pp->suspended == FALSE);

 771                 pp->suspended = TRUE;   /* prevent new transfers */

 773                 /*
 774                  * Wait if there's any activity on the port
 775                  */
 776                 if ((pp->e_busy == ECPP_BUSY) || (pp->e_busy == ECPP_FLUSH)) {
 777                         (void) cv_reltimedwait(&pp->pport_cv, &pp->umutex,
 778                             drv_sectohz(SUSPEND_TOUT),
 778                             SUSPEND_TOUT * drv_usectohz(1000000),
 779                             TR_CLOCK_TICK);
 780                         if ((pp->e_busy == ECPP_BUSY) ||
 781                             (pp->e_busy == ECPP_FLUSH)) {
 782                                 pp->suspended = FALSE;
 783                                 mutex_exit(&pp->umutex);
 784                                 ecpp_error(pp->dip,
 785                                     "ecpp_detach: suspend timeout\n");
 786                                 return (DDI_FAILURE);
 787                         }
 788                 }

 790                 mutex_exit(&pp->umutex);
 791                 return (DDI_SUCCESS);

 793         default:
 794                 return (DDI_FAILURE);
 795         }

 797         pp = ddi_get_soft_state(ecppsoft_statep, instance);
 798 #if defined(__x86)
 799         if (pp->hw == &x86 && pp->uh.x86.chn != 0xff)
 800                 (void) ddi_dmae_release(pp->dip, pp->uh.x86.chn);
 801 #endif
 802         if (pp->dma_handle != NULL)
 803                 ddi_dma_free_handle(&pp->dma_handle);

 805         ddi_remove_minor_node(dip, NULL);

 807         ddi_remove_softintr(pp->softintr_id);
```



```
 809         ddi_remove_intr(dip, (uint_t)0, pp->ecpp_trap_cookie);

 811         if (pp->ksp) {
 812                 kstat_delete(pp->ksp);
 813         }
 814         if (pp->intrstats) {
 815                 kstat_delete(pp->intrstats);
 816         }

 818         cv_destroy(&pp->pport_cv);

 820         mutex_destroy(&pp->umutex);

 822         ECPP_UNMAP_REGS(pp);

 824         kmem_free(pp->ioblock, IO_BLOCK_SZ);

 826         ddi_prop_remove_all(dip);

 828         ddi_soft_state_free(ecppsoft_statep, instance);

 830         return (DDI_SUCCESS);

 832 }
______unchanged_portion_omitted_

2710 static void
2711 ecpp_start(struct ecppunit *pp, caddr_t addr, size_t len)
2712 {
2713         ASSERT(mutex_owned(&pp->umutex));
2714         ASSERT(pp->e_busy == ECPP_BUSY);

2716         ecpp_error(pp->dip,
2717             "ecpp_start:current_mode=%x,current_phase=%x,ecr=%x,len=%d\n",
2718             pp->current_mode, pp->current_phase, ECR_READ(pp), len);

2720         pp->dma_dir = DDI_DMA_WRITE;    /* this is a forward transfer */

2722         switch (pp->current_mode) {
2723         case ECPP_NIBBLE_MODE:
2724                 (void) ecpp_1284_termination(pp);

2726                 /* After termination we are either Compatible or Centronics */

2728                 /* FALLTHRU */

2730         case ECPP_CENTRONICS:
2731         case ECPP_COMPAT_MODE:
2732                 if (pp->io_mode == ECPP_DMA) {
2733                         if (ecpp_init_dma_xfer(pp, addr, len) == FAILURE) {
2734                                 return;
2735                         }
2736                 } else {
2737                         /* PIO mode */
2738                         if (ecpp_prep_pio_xfer(pp, addr, len) == FAILURE) {
2739                                 return;
2740                         }
2741                         (void) ecpp_pio_writeb(pp);
2742                 }
2743                 break;

2745         case ECPP_DIAG_MODE: {
2746                 int     oldlen;

2748                 /* put superio into TFIFO mode, if not already */
2749                 ECR_WRITE(pp, ECPP_INTR_SRV | ECPP_INTR_MASK | ECR_mode_110);
```

```
2750                 /*
2751                  * DMA would block if the TFIFO is not empty
2752                  * if by this moment nobody read these bytes, they're gone
2753                  */
2754                 drv_usecwait(1);
2755                 if (!(ECR_READ(pp) & ECPP_FIFO_EMPTY)) {
2756                         ecpp_error(pp->dip,
2757                             "ecpp_start: TFIFO not empty, clearing\n");
2758                         ECR_WRITE(pp,
2759                             ECPP_INTR_SRV | ECPP_INTR_MASK | ECR_mode_001);
2760                         ECR_WRITE(pp,
2761                             ECPP_INTR_SRV | ECPP_INTR_MASK | ECR_mode_110);
2762                 }

2764                 /* we can DMA at most 16 bytes into TFIFO */
2765                 oldlen = len;
2766                 if (len > ECPP_FIFO_SZ) {
2767                         len = ECPP_FIFO_SZ;
2768                 }

2770                 if (ecpp_init_dma_xfer(pp, addr, len) == FAILURE) {
2771                         return;
2772                 }

2774                 /* put the rest of data back on the queue */
2775                 if (oldlen > len) {
2776                         ecpp_putback_untransfered(pp, addr + len, oldlen - len);
2777                 }

2779                 break;
2780         }

2782         case ECPP_ECP_MODE:
2783                 ASSERT(pp->current_phase == ECPP_PHASE_ECP_FWD_IDLE ||
2784                     pp->current_phase == ECPP_PHASE_ECP_REV_IDLE);

2786                 /* if in Reverse Phase negotiate to Forward */
2787                 if (pp->current_phase == ECPP_PHASE_ECP_REV_IDLE) {
2788                         if (ecp_reverse2forward(pp) == FAILURE) {
2789                                 if (pp->msg) {
2790                                         (void) putbq(pp->writeq, pp->msg);
2791                                 } else {
2792                                         ecpp_putback_untransfered(pp,
2793                                             addr, len);
2794                                 }
2795                         }
2796                 }

2798                 if (ecpp_init_dma_xfer(pp, addr, len) == FAILURE) {
2799                         return;
2800                 }

2802                 break;
2803         }

2805         /* schedule transfer timeout */
2806         pp->timeout_id = timeout(ecpp_xfer_timeout, (caddr_t)pp,
2807             drv_sectohz(pp->xfer_parms.write_timeout));
2807             pp->xfer_parms.write_timeout * drv_usectohz(1000000));
2808 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   84663 Wed Aug 19 07:24:59 2015
new/usr/src/uts/common/io/fdc.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1928 static int
1929 fdc_motorsm(struct fcu_obj *fjp, int input, int timeval)
1930 {
1931         struct fdcntlr *fcp = fjp->fj_fdc;
1932         int unit = fjp->fj_unit & 3;
1933         int old_mstate;
1934         int rval = 0;
1935         uchar_t motorbit;

1937         ASSERT(MUTEX_HELD(&fcp->c_dorlock));
1938         old_mstate = fcp->c_mtrstate[unit];
1939         encode(motor_onbits, unit, &motorbit);

1941         switch (input) {
1942         case FMI_TIMER:         /* timer expired */
1943                 fcp->c_motort[unit] = 0;
1944                 switch (old_mstate) {
1945                 case FMS_START:
1946                 case FMS_DELAY:
1947                         fcp->c_mtrstate[unit] = FMS_ON;
1948                         break;
1949                 case FMS_KILLST:
1950                         fcp->c_motort[unit] = timeout(fdmotort, (void *)fjp,
1951                             drv_sectohz(1));
1951                             drv_usectohz(1000000));
1952                         fcp->c_mtrstate[unit] = FMS_IDLE;
1953                         break;
1954                 case FMS_IDLE:
1955                         fcp->c_digout &= ~motorbit;
1956                         outb(fcp->c_regbase + FCR_DOR, fcp->c_digout);
1957                         fcp->c_mtrstate[unit] = FMS_OFF;
1958                         fjp->fj_flags &= ~FUNIT_3DMODE;
1959                         break;
1960                 case 86:
1961                         rval = -1;
1962                         break;
1963                 case FMS_OFF:
1964                 case FMS_ON:
1965                 default:
1966                         rval = -2;
1967                 }
1968                 break;

1970         case FMI_STARTCMD:      /* start command */
1971                 switch (old_mstate) {
1972                 case FMS_IDLE:
1973                         fcp->c_mtrstate[unit] = 86;
1974                         mutex_exit(&fcp->c_dorlock);
1975                         (void) untimeout(fcp->c_motort[unit]);
1976                         mutex_enter(&fcp->c_dorlock);
1977                         fcp->c_motort[unit] = 0;
1978                         fcp->c_mtrstate[unit] = FMS_ON;
1979                         break;
1980                 case FMS_OFF:
1981                         fcp->c_digout |= motorbit;
1982                         outb(fcp->c_regbase + FCR_DOR, fcp->c_digout);

1984                         /* start motor_spinup_timer */
1985                         ASSERT(timeval > 0);
```



```
1986                         fcp->c_motort[unit] = timeout(fdmotort,  (void *)fjp,
1987                             drv_usectohz(100000 * timeval));
1988                         /* FALLTHROUGH */
1989                 case FMS_KILLST:
1990                         fcp->c_mtrstate[unit] = FMS_START;
1991                         break;
1992                 default:
1993                         rval = -2;
1994                 }
1995                 break;

1997         case FMI_RSTARTCMD:     /* restart command */
1998                 if (fcp->c_motort[unit] != 0) {
1999                         fcp->c_mtrstate[unit] = 86;
2000                         mutex_exit(&fcp->c_dorlock);
2001                         (void) untimeout(fcp->c_motort[unit]);
2002                         mutex_enter(&fcp->c_dorlock);
2003                 }
2004                 ASSERT(timeval > 0);
2005                 fcp->c_motort[unit] = timeout(fdmotort, (void *)fjp,
2006                     drv_usectohz(100000 * timeval));
2007                 fcp->c_mtrstate[unit] = FMS_START;
2008                 break;

2010         case FMI_DELAYCMD:      /* delay command */
2011                 if (fcp->c_motort[unit] == 0)
2012                         fcp->c_motort[unit] = timeout(fdmotort,  (void *)fjp,
2013                             drv_usectohz(15000));
2014                 fcp->c_mtrstate[unit] = FMS_DELAY;
2015                 break;

2017         case FMI_IDLECMD:       /* idle command */
2018                 switch (old_mstate) {
2019                 case FMS_DELAY:
2020                         fcp->c_mtrstate[unit] = 86;
2021                         mutex_exit(&fcp->c_dorlock);
2022                         (void) untimeout(fcp->c_motort[unit]);
2023                         mutex_enter(&fcp->c_dorlock);
2024                         /* FALLTHROUGH */
2025                 case FMS_ON:
2026                         ASSERT(timeval > 0);
2027                         fcp->c_motort[unit] = timeout(fdmotort, (void *)fjp,
2028                             drv_usectohz(100000 * timeval));
2029                         fcp->c_mtrstate[unit] = FMS_IDLE;
2030                         break;
2031                 case FMS_START:
2032                         fcp->c_mtrstate[unit] = FMS_KILLST;
2033                         break;
2034                 default:
2035                         rval = -2;
2036                 }
2037                 break;

2039         default:
2040                 rval = -3;
2041         }
2042         if (rval) {
2043                 FCERRPRINT(FDEP_L4, FDEM_EXEC, (CE_WARN,
2044                     "fdc_motorsm: unit %d  bad input %d or bad state %d",
2045                     (int)fjp->fj_unit, input, old_mstate));
2046 #if 0
2047                 cmn_err(CE_WARN,
2048                     "fdc_motorsm: unit %d  bad input %d or bad state %d",
2049                     (int)fjp->fj_unit, input, old_mstate);
2050                 fcp->c_mtrstate[unit] = FMS_OFF;
2051                 if (fcp->c_motort[unit] != 0) {
```

```
2052                        mutex_exit(&fcp->c_dorlock);
2053                        (void) untimeout(fcp->c_motort[unit]);
2054                        mutex_enter(&fcp->c_dorlock);
2055                        fcp->c_motort[unit] = 0;
2056                }
2057 #endif
2058        } else
2059                FCERRPRINT(FDEP_L0, FDEM_EXEC,
2060                    (CE_CONT, "fdc_motorsm unit %d: input %d,  %d -> %d\n",
2061                    (int)fjp->fj_unit, input, old_mstate,
2062                    fcp->c_mtrstate[unit]));
2063        return (rval);
2064 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31594 Wed Aug 19 07:24:59 2015
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_clock.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at
 * http://www.opensource.org/licenses/cddl1.txt.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2004-2012 Emulex. All rights reserved.
 * Use is subject to license terms.
 */

#include <emlxs.h>

/* Timer period in seconds */
#define EMLXS_TIMER_PERIOD              1       /* secs */
#define EMLXS_PKT_PERIOD                5       /* secs */
#define EMLXS_UB_PERIOD                 60      /* secs */

EMLXS_MSG_DEF(EMLXS_CLOCK_C);


static void emlxs_timer_check_loopback(emlxs_hba_t *hba);

#ifdef DHCHAP_SUPPORT
static void emlxs_timer_check_dhchap(emlxs_port_t *port);
#endif /* DHCHAP_SUPPORT */

static void     emlxs_timer_check_pools(emlxs_hba_t *hba);
static void     emlxs_timer(void *arg);
static void     emlxs_timer_check_fw_update(emlxs_hba_t *hba);
static void     emlxs_timer_check_heartbeat(emlxs_hba_t *hba);
static uint32_t emlxs_timer_check_pkts(emlxs_hba_t *hba, uint8_t *flag);
static void     emlxs_timer_check_nodes(emlxs_port_t *port, uint8_t *flag);
static void     emlxs_timer_check_linkup(emlxs_hba_t *hba);
static void     emlxs_timer_check_discovery(emlxs_port_t *port);
static void     emlxs_timer_check_clean_address(emlxs_port_t *port);
static void     emlxs_timer_check_ub(emlxs_port_t *port);
static void     emlxs_timer_check_channels(emlxs_hba_t *hba, uint8_t *flag);
static uint32_t emlxs_pkt_chip_timeout(emlxs_port_t *port, emlxs_buf_t *sbp,
                        Q *abortq, uint8_t *flag);

#ifdef TX_WATCHDOG
static void     emlxs_tx_watchdog(emlxs_hba_t *hba);
#endif /* TX_WATCHDOG */

extern clock_t
```



```
emlxs_timeout(emlxs_hba_t *hba, uint32_t timeout)
{
        emlxs_config_t *cfg = &CFG;
        clock_t time;

        /* Set thread timeout */
        if (cfg[CFG_TIMEOUT_ENABLE].current) {
                (void) drv_getparm(LBOLT, &time);
                time += drv_sectohz(timeout);
                time += (timeout * drv_usectohz(1000000));
        } else {
                time = -1;
        }

        return (time);

} /* emlxs_timeout() */


static void
emlxs_timer(void *arg)
{
        emlxs_hba_t *hba = (emlxs_hba_t *)arg;
        emlxs_port_t *port = &PPORT;

        if (!hba->timer_id) {
                return;
        }

        mutex_enter(&EMLXS_TIMER_LOCK);

        /* Only one timer thread is allowed */
        if (hba->timer_flags & EMLXS_TIMER_BUSY) {
                mutex_exit(&EMLXS_TIMER_LOCK);
                return;
        }

        /* Check if a kill request has been made */
        if (hba->timer_flags & EMLXS_TIMER_KILL) {
                hba->timer_id = 0;
                hba->timer_tics = 0;
                hba->timer_flags |= EMLXS_TIMER_ENDED;

                mutex_exit(&EMLXS_TIMER_LOCK);
                return;
        }

        hba->timer_flags |= (EMLXS_TIMER_BUSY | EMLXS_TIMER_STARTED);
        hba->timer_tics = DRV_TIME;

        /* Check io_active count (Safety net) */
        if (hba->io_active & 0x80000000) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_debug_msg,
                    "Timer: io_active=0x%x. Reset to zero.", hba->io_active);
                hba->io_active = 0;
        }

        mutex_exit(&EMLXS_TIMER_LOCK);

        EMLXS_SLI_POLL_ERRATT(hba);

        /* Perform standard checks */
        emlxs_timer_checks(hba);

        /* Restart the timer */
        mutex_enter(&EMLXS_TIMER_LOCK);
```

```
128         hba->timer_flags &= ~EMLXS_TIMER_BUSY;

130         /* If timer is still enabled, restart it */
131         if (!(hba->timer_flags & EMLXS_TIMER_KILL)) {
132                 hba->timer_id =
133                     timeout(emlxs_timer, (void *)hba,
134                     drv_sectohz(EMLXS_TIMER_PERIOD));
134                     (EMLXS_TIMER_PERIOD * drv_usectohz(1000000)));
135         } else {
136                 hba->timer_id = 0;
137                 hba->timer_flags |= EMLXS_TIMER_ENDED;
138         }

140         mutex_exit(&EMLXS_TIMER_LOCK);

142         return;

144 } /* emlxs_timer() */
```

*unchanged_portion_omitted_*

```
231 extern void
232 emlxs_timer_start(emlxs_hba_t *hba)
233 {
234         if (hba->timer_id) {
235                 return;
236         }

238         /* Restart the timer */
239         mutex_enter(&EMLXS_TIMER_LOCK);
240         if (!hba->timer_id) {
241                 hba->timer_flags = 0;
242                 hba->timer_id =
243                     timeout(emlxs_timer, (void *)hba, drv_sectohz(1));
243                     timeout(emlxs_timer, (void *)hba, drv_usectohz(1000000));
244         }
245         mutex_exit(&EMLXS_TIMER_LOCK);

247 } /* emlxs_timer_start() */
```

*unchanged_portion_omitted_*

```
**********************************************************
  276537 Wed Aug 19 07:24:59 2015
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_dfc.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


4355 extern uint32_t
4356 emlxs_set_hba_mode(emlxs_hba_t *hba, uint32_t mode)
4357 {
4358         emlxs_port_t    *port = &PPORT;
4359         uint32_t        i;

4361         mutex_enter(&EMLXS_PORT_LOCK);

4363         /* Wait if adapter is in transition */
4364         i = 0;
4365         while ((hba->flag & (FC_ONLINING_MODE | FC_OFFLINING_MODE))) {
4366                 if (i++ > 30) {
4367                         break;
4368                 }

4370                 mutex_exit(&EMLXS_PORT_LOCK);
4371                 delay(drv_sectohz(1));
4371                 delay(drv_usectohz(1000000));
4372                 mutex_enter(&EMLXS_PORT_LOCK);
4373         }

4375         if (hba->sli_mode <= EMLXS_HBA_SLI3_MODE) {
4376                 switch (mode) {
4377                 case DDI_SHOW:
4378                         break;

4380                 case DDI_ONDI:
4381                         if (hba->flag & FC_OFFLINE_MODE) {
4382                                 mutex_exit(&EMLXS_PORT_LOCK);
4383                                 (void) emlxs_online(hba);
4384                                 mutex_enter(&EMLXS_PORT_LOCK);
4385                         }
4386                         break;


4389                 /* Killed + Restart state */
4390                 case DDI_OFFDI:
4391                         if (hba->flag & FC_ONLINE_MODE) {
4392                                 mutex_exit(&EMLXS_PORT_LOCK);

4394                                 (void) emlxs_offline(hba, 0);

4396                                 /* Reset with restart */
4397                                 EMLXS_SLI_HBA_RESET(hba, 1, 1, 0);

4399                                 mutex_enter(&EMLXS_PORT_LOCK);
4400                         } else if (hba->state < FC_INIT_START) {
4401                                 mutex_exit(&EMLXS_PORT_LOCK);

4403                                 /* Reset with restart */
4404                                 EMLXS_SLI_HBA_RESET(hba, 1, 1, 0);

4406                                 mutex_enter(&EMLXS_PORT_LOCK);
4407                         }

4409                         break;

4411                 /* Killed + Reset state */
```



```
4412                 case DDI_WARMDI:
4413                         if (hba->flag & FC_ONLINE_MODE) {
4414                                 mutex_exit(&EMLXS_PORT_LOCK);

4416                                 (void) emlxs_offline(hba, 0);

4418                                 /* Reset with no restart */
4419                                 EMLXS_SLI_HBA_RESET(hba, 0, 0, 0);

4421                                 mutex_enter(&EMLXS_PORT_LOCK);
4422                         } else if (hba->state != FC_WARM_START) {
4423                                 mutex_exit(&EMLXS_PORT_LOCK);

4425                                 /* Reset with no restart */
4426                                 EMLXS_SLI_HBA_RESET(hba, 0, 0, 0);

4428                                 mutex_enter(&EMLXS_PORT_LOCK);
4429                         }

4431                         break;

4433                 /* Killed */
4434                 case DDI_DIAGDI:
4435                         if (hba->flag & FC_ONLINE_MODE) {
4436                                 mutex_exit(&EMLXS_PORT_LOCK);

4438                                 (void) emlxs_offline(hba, 0);

4440                                 mutex_enter(&EMLXS_PORT_LOCK);
4441                         } else if (hba->state != FC_KILLED) {
4442                                 mutex_exit(&EMLXS_PORT_LOCK);

4444                                 EMLXS_SLI_HBA_KILL(hba);

4446                                 mutex_enter(&EMLXS_PORT_LOCK);
4447                         }

4449                         break;

4451                 default:
4452                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
4453                             "set_hba_mode: Invalid mode. mode=%x", mode);
4454                         mutex_exit(&EMLXS_PORT_LOCK);
4455                         return (0);
4456                 }

4458                 /* Wait if adapter is in transition */
4459                 i = 0;
4460                 while ((hba->flag & (FC_ONLINING_MODE | FC_OFFLINING_MODE))) {
4461                         if (i++ > 30) {
4462                                 break;
4463                         }

4465                         mutex_exit(&EMLXS_PORT_LOCK);
4466                         delay(drv_sectohz(1));
4466                         delay(drv_usectohz(1000000));
4467                         mutex_enter(&EMLXS_PORT_LOCK);
4468                 }

4470                 /* Return current state */
4471                 if (hba->flag & FC_ONLINE_MODE) {
4472                         mode = DDI_ONDI;
4473                 } else if (hba->state == FC_KILLED) {
4474                         mode = DDI_DIAGDI;
4475                 } else if (hba->state == FC_WARM_START) {
4476                         mode = DDI_WARMDI;
```

```
4477                } else {
4478                        mode = DDI_OFFDI;
4479                }

4481                mutex_exit(&EMLXS_PORT_LOCK);

4483                return (mode);

4485        } else { /* SLI4 */
4486                switch (mode) {
4487                case DDI_SHOW:
4488                        break;

4490                case DDI_ONDI:
4491                        if (hba->flag & FC_OFFLINE_MODE) {
4492                                mutex_exit(&EMLXS_PORT_LOCK);
4493                                (void) emlxs_online(hba);
4494                                mutex_enter(&EMLXS_PORT_LOCK);
4495                        }
4496                        break;

4498                case DDI_OFFDI:
4499                        if (hba->flag & FC_ONLINE_MODE) {
4500                                mutex_exit(&EMLXS_PORT_LOCK);

4502                                (void) emlxs_offline(hba, 0);

4504                                /* Reset with restart */
4505                                EMLXS_SLI_HBA_RESET(hba, 1, 1, 0);

4507                                mutex_enter(&EMLXS_PORT_LOCK);
4508                        } else if (hba->state < FC_INIT_START) {
4509                                mutex_exit(&EMLXS_PORT_LOCK);

4511                                /* Reset with restart */
4512                                EMLXS_SLI_HBA_RESET(hba, 1, 1, 0);

4514                                mutex_enter(&EMLXS_PORT_LOCK);
4515                        }
4516                        break;

4518                case DDI_DIAGDI:
4519                        if (!(hba->model_info.chip & EMLXS_LANCER_CHIP)) {
4520                                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
4521                                    "set_hba_mode: Invalid mode. mode=%x",
4522                                    mode);
4523                                mutex_exit(&EMLXS_PORT_LOCK);
4524                                return (0);
4525                        }

4527                        mutex_exit(&EMLXS_PORT_LOCK);
4528                        (void) emlxs_reset(port,
4529                            EMLXS_DFC_RESET_ALL_FORCE_DUMP);

4531                        return (mode);

4533                case DDI_WARMDI:
4534                default:
4535                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
4536                            "set_hba_mode: Invalid mode. mode=%x", mode);
4537                        mutex_exit(&EMLXS_PORT_LOCK);
4538                        return (0);
4539                }

4541                /* Wait if adapter is in transition */
4542                i = 0;
```



```
4543                while ((hba->flag & (FC_ONLINING_MODE | FC_OFFLINING_MODE))) {
4544                        if (i++ > 30) {
4545                                break;
4546                        }

4548                        mutex_exit(&EMLXS_PORT_LOCK);
4549                        delay(drv_sectohz(1));
4549                        delay(drv_usectohz(1000000));
4550                        mutex_enter(&EMLXS_PORT_LOCK);
4551                }

4553                /* Return current state */
4554                if (hba->flag & FC_ONLINE_MODE) {
4555                        mode = DDI_ONDI;
4556                } else {
4557                        mode = DDI_OFFDI;
4558                }

4560                mutex_exit(&EMLXS_PORT_LOCK);

4562                return (mode);
4563        }

4565 } /* emlxs_set_hba_mode() */
```

_____***unchanged_portion_omitted_***

```
8168 /*ARGSUSED*/
8169 static int32_t
8170 emlxs_dfc_set_menlo_loopback(emlxs_hba_t *hba)
8171 {
8172        emlxs_port_t *port = &PPORT;
8173        MAILBOXQ *mbq = NULL;
8174        MAILBOX *mb = NULL;
8175        fc_packet_t *pkt = NULL;
8176        uint32_t mbxstatus;
8177        uint32_t i;
8178        uint32_t offset;
8179        uint32_t rval = 0;
8180        menlo_cmd_t *cmd;

8182        mbq = (MAILBOXQ *)kmem_zalloc(sizeof (MAILBOXQ),
8183            KM_SLEEP);

8185        mb = (MAILBOX *)mbq;

8187        /* SET MENLO maint mode */
8188        /* Create the set_variable mailbox request */
8189        emlxs_mb_set_var(hba, mbq, 0x103107, 1);

8191        mbq->flag |= MBQ_PASSTHRU;

8193        /* issue the mbox cmd to the sli */
8194        mbxstatus = EMLXS_SLI_ISSUE_MBOX_CMD(hba, mbq, MBX_WAIT, 0);

8196        if (mbxstatus) {
8197                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8198                    "%s: %s failed. mbxstatus=0x%x",
8199                    emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE),
8200                    emlxs_mb_cmd_xlate(mb->mbxCommand), mbxstatus);

8202                rval = DFC_IO_ERROR;
8203                if (mbxstatus == MBX_TIMEOUT)
8204                        rval = DFC_TIMEOUT;
8205                goto done;
8206        }
```

```
8209         /* Wait 30 sec for maint mode */
8210         i = 0;
8211         do {
8212                 if (i++ > 300) {
8213                         break;
8214                 }

8216                 delay(drv_usectohz(100000));

8218         } while (!(hba->flag & FC_MENLO_MODE));

8220         if (!(hba->flag & FC_MENLO_MODE)) {
8221                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8222                     "%s: Unable to enter maint mode.",
8223                     emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));

8225                 rval = DFC_DRV_ERROR;
8226                 goto done;
8227         }

8229         offset = emlxs_dfc_menlo_port_offset(hba);
8230         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8231             "%s: Entered maint mode. Port offset: %d",
8232             emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE), offset);


8235         /* Issue Menlo loopback command */
8236         if (!(pkt = emlxs_pkt_alloc(port, sizeof (menlo_cmd_t),
8237             sizeof (uint32_t), 0, KM_NOSLEEP))) {
8238                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8239                     "%s: Unable to allocate packet.",
8240                     emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));

8242                 rval = DFC_SYSRES_ERROR;
8243                 goto done;
8244         }

8246         /* Make this a polled IO */
8247         pkt->pkt_tran_flags &= ~FC_TRAN_INTR;
8248         pkt->pkt_tran_flags |= FC_TRAN_NO_INTR;
8249         pkt->pkt_comp = NULL;
8250         pkt->pkt_tran_type = FC_PKT_EXCHANGE;
8251         pkt->pkt_timeout = 30;

8253         /* Build the fc header */
8254         pkt->pkt_cmd_fhdr.d_id = LE_SWAP24_LO(EMLXS_MENLO_DID);
8255         pkt->pkt_cmd_fhdr.r_ctl = R_CTL_COMMAND;
8256         pkt->pkt_cmd_fhdr.s_id = LE_SWAP24_LO(port->did);
8257         pkt->pkt_cmd_fhdr.type = EMLXS_MENLO_TYPE;
8258         pkt->pkt_cmd_fhdr.f_ctl =
8259             F_CTL_FIRST_SEQ | F_CTL_END_SEQ | F_CTL_SEQ_INITIATIVE;
8260         pkt->pkt_cmd_fhdr.seq_id = 0;
8261         pkt->pkt_cmd_fhdr.df_ctl = 0;
8262         pkt->pkt_cmd_fhdr.seq_cnt = 0;
8263         pkt->pkt_cmd_fhdr.ox_id = 0xFFFF;
8264         pkt->pkt_cmd_fhdr.rx_id = 0xFFFF;
8265         pkt->pkt_cmd_fhdr.ro = 0;

8267         cmd = (menlo_cmd_t *)pkt->pkt_cmd;
8268         cmd->code = BE_SWAP32(MENLO_CMD_LOOPBACK);
8269         cmd->lb.context = BE_SWAP32(offset);
8270         cmd->lb.type = BE_SWAP32(MENLO_LOOPBACK_ENABLE);

8272         if (emlxs_pkt_send(pkt, 1) != FC_SUCCESS) {
8273                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
```



```
8274                     "%s: Unable to send packet.",
8275                     emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));

8277                 rval = DFC_IO_ERROR;
8278                 goto done;
8279         }

8281         if (pkt->pkt_state != FC_PKT_SUCCESS) {
8282                 if (pkt->pkt_state == FC_PKT_TIMEOUT) {
8283                         EMLXS_MSGF(EMLXS_CONTEXT,
8284                             &emlxs_dfc_error_msg,
8285                             "%s: Pkt Transport error. Pkt Timeout.",
8286                             emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));
8287                         rval = DFC_TIMEOUT;
8288                 } else if ((pkt->pkt_state == FC_PKT_LOCAL_RJT) &&
8289                     (pkt->pkt_reason == FC_REASON_OVERRUN)) {
8290                         EMLXS_MSGF(EMLXS_CONTEXT,
8291                             &emlxs_dfc_error_msg,
8292                             "%s: Pkt Transport error. Rsp overrun.",
8293                             emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));
8294                         rval = DFC_RSP_BUF_OVERRUN;
8295                 } else {
8296                         EMLXS_MSGF(EMLXS_CONTEXT,
8297                             &emlxs_dfc_error_msg,
8298                             "%s: Pkt Transport error. state=%x",
8299                             emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE),
8300                             pkt->pkt_state);
8301                         rval = DFC_IO_ERROR;
8302                 }
8303                 goto done;
8304         }


8307         /* CLEAR MENLO maint mode */
8308         /* Create the set_variable mailbox request */
8309         emlxs_mb_set_var(hba, mbq, 0x103107, 0);

8311         mbq->flag |= MBQ_PASSTHRU;

8313         /* issue the mbox cmd to the sli */
8314         mbxstatus = EMLXS_SLI_ISSUE_MBOX_CMD(hba, mbq, MBX_WAIT, 0);

8316         if (mbxstatus) {
8317                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8318                     "%s: %s failed. mbxstatus=0x%x",
8319                     emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE),
8320                     emlxs_mb_cmd_xlate(mb->mbxCommand), mbxstatus);

8322                 rval = DFC_IO_ERROR;
8323                 if (mbxstatus == MBX_TIMEOUT)
8324                         rval = DFC_TIMEOUT;
8325         }

8327         delay(drv_sectohz(1));
8327         delay(drv_usectohz(1000000));
8328         i = 0;
8329         while ((hba->state < FC_LINK_UP) && (hba->state != FC_ERROR)) {
8330                 delay(drv_usectohz(100000));
8331                 i++;

8333                 if (i == 300) {
8334                         rval = DFC_TIMEOUT;

8336                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
8337                             "%s: Linkup timeout.",
8338                             emlxs_dfc_xlate(EMLXS_LOOPBACK_MODE));
```

```
8340                        goto done;
8341                }
8342        }

8344 done:
8345        /* Free allocated mbox memory */
8346        if (mbq) {
8347                kmem_free(mbq, sizeof (MAILBOXQ));
8348        }
8349        if (pkt) {
8350                emlxs_pkt_free(pkt);
8351        }
8352        return (rval);
8353 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  100295 Wed Aug 19 07:25:00 2015
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_fcp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


1922 extern int
1923 emlxs_offline(emlxs_hba_t *hba, uint32_t reset_requested)
1924 {
1925         emlxs_port_t *port = &PPORT;
1926         uint32_t i = 0;
1927         int rval = 1;

1929         /* Make sure adapter is online or exit trying (30 seconds) */
1930         while (i++ < 30) {
1931                 /* Check if adapter is already going offline */
1932                 if (hba->flag & (FC_OFFLINE_MODE | FC_OFFLINING_MODE)) {
1933                         return (0);
1934                 }

1936                 mutex_enter(&EMLXS_PORT_LOCK);

1938                 /* Check again */
1939                 if (hba->flag & (FC_OFFLINE_MODE | FC_OFFLINING_MODE)) {
1940                         mutex_exit(&EMLXS_PORT_LOCK);
1941                         return (0);
1942                 }

1944                 /* Check if adapter is online */
1945                 if (hba->flag & FC_ONLINE_MODE) {
1946                         /* Mark it going offline */
1947                         hba->flag &= ~FC_ONLINE_MODE;
1948                         hba->flag |= FC_OFFLINING_MODE;

1950                         /* Currently !FC_ONLINE_MODE and !FC_OFFLINE_MODE */
1951                         mutex_exit(&EMLXS_PORT_LOCK);
1952                         break;
1953                 }

1955                 mutex_exit(&EMLXS_PORT_LOCK);

1957                 BUSYWAIT_MS(1000);
1958         }

1960         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_adapter_trans_msg,
1961             "Going offline...");

1963         /* Declare link down */
1964         if (hba->sli_mode == EMLXS_HBA_SLI4_MODE) {
1965                 (void) emlxs_fcf_shutdown_notify(port, 1);
1966         } else {
1967                 emlxs_linkdown(hba);
1968         }

1970 #ifdef SFCT_SUPPORT
1971         if (port->flag & EMLXS_TGT_ENABLED) {
1972                 (void) emlxs_fct_port_shutdown(port);
1973         }
1974 #endif /* SFCT_SUPPORT */

1976         /* Check if adapter was shutdown */
1977         if (hba->flag & FC_HARDWARE_ERROR) {
1978                 /*
1979                  * Force mailbox cleanup
```



```
1980                  * This will wake any sleeping or polling threads
1981                  */
1982                 emlxs_mb_fini(hba, NULL, MBX_HARDWARE_ERROR);
1983         }

1985         /* Pause here for the IO to settle */
1986         delay(drv_sectohz(1));
1986         delay(drv_usectohz(1000000));   /* 1 sec */

1988         /* Unregister all nodes */
1989         emlxs_ffcleanup(hba);

1991         if (hba->bus_type == SBUS_FC) {
1992                 WRITE_SBUS_CSR_REG(hba, FC_SHS_REG(hba), 0x9A);
1993 #ifdef FMA_SUPPORT
1994                 /* Access handle validation */
1995                 EMLXS_CHK_ACC_HANDLE(hba, hba->sli.sli3.sbus_csr_handle);
1996 #endif  /* FMA_SUPPORT */
1997         }

1999         /* Stop the timer */
2000         emlxs_timer_stop(hba);

2002         /* For safety flush every iotag list */
2003         if (emlxs_iotag_flush(hba)) {
2004                 /* Pause here for the IO to flush */
2005                 delay(drv_usectohz(1000));
2006         }

2008         /* Wait for poll command request to settle */
2009         while (hba->io_poll_count > 0) {
2010                 delay(drv_usectohz(2000000));   /* 2 sec */
2011         }

2013         /* Shutdown the adapter interface */
2014         EMLXS_SLI_OFFLINE(hba, reset_requested);

2016         mutex_enter(&EMLXS_PORT_LOCK);
2017         hba->flag |= FC_OFFLINE_MODE;
2018         hba->flag &= ~FC_OFFLINING_MODE;
2019         mutex_exit(&EMLXS_PORT_LOCK);

2021         rval = 0;

2023         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_offline_msg, NULL);

2025 done:

2027         return (rval);

2029 } /* emlxs_offline() */
______unchanged_portion_omitted_
```

```
**********************************************************
  298461 Wed Aug 19 07:25:00 2015
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_solaris.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_



2619 static void
2620 emlxs_poll(emlxs_port_t *port, emlxs_buf_t *sbp)
2621 {
2622         emlxs_hba_t     *hba = HBA;
2623         fc_packet_t     *pkt = PRIV2PKT(sbp);
2624         clock_t         timeout;
2625         clock_t         time;
2626         CHANNEL *cp;
2627         int             in_panic = 0;

2629         mutex_enter(&EMLXS_PORT_LOCK);
2630         hba->io_poll_count++;
2631         mutex_exit(&EMLXS_PORT_LOCK);

2633         /* Check for panic situation */
2634         cp = (CHANNEL *)sbp->channel;

2636         if (ddi_in_panic()) {
2637                 in_panic = 1;
2638                 /*
2639                  * In panic situations there will be one thread with
2640                  * no interrrupts (hard or soft) and no timers
2641                  */

2643                 /*
2644                  * We must manually poll everything in this thread
2645                  * to keep the driver going.
2646                  */

2648                 /* Keep polling the chip until our IO is completed */
2649                 /* Driver's timer will not function during panics. */
2650                 /* Therefore, timer checks must be performed manually. */
2651                 (void) drv_getparm(LBOLT, &time);
2652                 timeout = time + drv_sectohz(1);
2652                 timeout = time + drv_usectohz(1000000);
2653                 while (!(sbp->pkt_flags & PACKET_COMPLETED)) {
2654                         EMLXS_SLI_POLL_INTR(hba);
2655                         (void) drv_getparm(LBOLT, &time);

2657                         /* Trigger timer checks periodically */
2658                         if (time >= timeout) {
2659                                 emlxs_timer_checks(hba);
2660                                 timeout = time + drv_sectohz(1);
2660                                 timeout = time + drv_usectohz(1000000);
2661                         }
2662                 }
2663         } else {
2664                 /* Wait for IO completion */
2665                 /* The driver's timer will detect */
2666                 /* any timeout and abort the I/O. */
2667                 mutex_enter(&EMLXS_PKT_LOCK);
2668                 while (!(sbp->pkt_flags & PACKET_COMPLETED)) {
2669                         cv_wait(&EMLXS_PKT_CV, &EMLXS_PKT_LOCK);
2670                 }
2671                 mutex_exit(&EMLXS_PKT_LOCK);
2672         }
```



```
2674         /* Check for fcp reset pkt */
2675         if (sbp->pkt_flags & PACKET_FCP_RESET) {
2676                 if (sbp->pkt_flags & PACKET_FCP_TGT_RESET) {
2677                         /* Flush the IO's on the chipq */
2678                         (void) emlxs_chipq_node_flush(port,
2679                             &hba->chan[hba->channel_fcp],
2680                             sbp->node, sbp);
2681                 } else {
2682                         /* Flush the IO's on the chipq for this lun */
2683                         (void) emlxs_chipq_lun_flush(port,
2684                             sbp->node, sbp->lun, sbp);
2685                 }

2687                 if (sbp->flush_count == 0) {
2688                         emlxs_node_open(port, sbp->node, hba->channel_fcp);
2689                         goto done;
2690                 }

2692                 /* Set the timeout so the flush has time to complete */
2693                 timeout = emlxs_timeout(hba, 60);
2694                 (void) drv_getparm(LBOLT, &time);
2695                 while ((time < timeout) && sbp->flush_count > 0) {
2696                         delay(drv_usectohz(500000));
2697                         (void) drv_getparm(LBOLT, &time);
2698                 }

2700                 if (sbp->flush_count == 0) {
2701                         emlxs_node_open(port, sbp->node, hba->channel_fcp);
2702                         goto done;
2703                 }

2705                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_flush_timeout_msg,
2706                     "sbp=%p flush_count=%d. Waiting...", sbp,
2707                     sbp->flush_count);

2709                 /* Let's try this one more time */

2711                 if (sbp->pkt_flags & PACKET_FCP_TGT_RESET) {
2712                         /* Flush the IO's on the chipq */
2713                         (void) emlxs_chipq_node_flush(port,
2714                             &hba->chan[hba->channel_fcp],
2715                             sbp->node, sbp);
2716                 } else {
2717                         /* Flush the IO's on the chipq for this lun */
2718                         (void) emlxs_chipq_lun_flush(port,
2719                             sbp->node, sbp->lun, sbp);
2720                 }

2722                 /* Reset the timeout so the flush has time to complete */
2723                 timeout = emlxs_timeout(hba, 60);
2724                 (void) drv_getparm(LBOLT, &time);
2725                 while ((time < timeout) && sbp->flush_count > 0) {
2726                         delay(drv_usectohz(500000));
2727                         (void) drv_getparm(LBOLT, &time);
2728                 }

2730                 if (sbp->flush_count == 0) {
2731                         emlxs_node_open(port, sbp->node, hba->channel_fcp);
2732                         goto done;
2733                 }

2735                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_flush_timeout_msg,
2736                     "sbp=%p flush_count=%d. Resetting link.", sbp,
2737                     sbp->flush_count);

2739                 /* Let's first try to reset the link */
```

```
2740                 (void) emlxs_reset(port, FC_FCA_LINK_RESET);

2742                 if (sbp->flush_count == 0) {
2743                         goto done;
2744                 }

2746                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_flush_timeout_msg,
2747                     "sbp=%p flush_count=%d. Resetting HBA.", sbp,
2748                     sbp->flush_count);

2750                 /* If that doesn't work, reset the adapter */
2751                 (void) emlxs_reset(port, FC_FCA_RESET);

2753                 if (sbp->flush_count != 0) {
2754                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_flush_timeout_msg,
2755                             "sbp=%p flush_count=%d. Giving up.", sbp,
2756                             sbp->flush_count);
2757                 }

2759         }
2760         /* PACKET_FCP_RESET */
2761 done:

2763         /* Packet has been declared completed and is now ready to be returned */

2765 #if (EMLXS_MODREVX == EMLXS_MODREV2X)
2766         emlxs_unswap_pkt(sbp);
2767 #endif  /* EMLXS_MODREV2X */

2769         mutex_enter(&sbp->mtx);
2770         sbp->pkt_flags |= PACKET_ULP_OWNED;
2771         mutex_exit(&sbp->mtx);

2773         mutex_enter(&EMLXS_PORT_LOCK);
2774         hba->io_poll_count--;
2775         mutex_exit(&EMLXS_PORT_LOCK);

2777 #ifdef FMA_SUPPORT
2778         if (!in_panic) {
2779                 emlxs_check_dma(hba, sbp);
2780         }
2781 #endif

2783         /* Make ULP completion callback if required */
2784         if (pkt->pkt_comp) {
2785                 cp->ulpCmplCmd++;
2786                 (*pkt->pkt_comp) (pkt);
2787         }

2789 #ifdef FMA_SUPPORT
2790         if (hba->flag & FC_DMA_CHECK_ERROR) {
2791                 emlxs_thread_spawn(hba, emlxs_restart_thread,
2792                     NULL, NULL);
2793         }
2794 #endif

2796         return;

2798 } /* emlxs_poll() */
```

_____unchanged_portion_omitted_

```
3412 /*ARGSUSED*/
3413 extern int
3414 emlxs_fca_pkt_abort(opaque_t fca_port_handle, fc_packet_t *pkt, int32_t sleep)
3415 {
```



```
3416         emlxs_port_t    *port = (emlxs_port_t *)fca_port_handle;
3417         emlxs_hba_t     *hba = HBA;
3418         emlxs_config_t  *cfg = &CFG;

3420         emlxs_buf_t     *sbp;
3421         NODELIST        *nlp;
3422         NODELIST        *prev_nlp;
3423         uint8_t         channelno;
3424         CHANNEL *cp;
3425         clock_t         pkt_timeout;
3426         clock_t         timer;
3427         clock_t         time;
3428         int32_t         pkt_ret;
3429         IOCBQ           *iocbq;
3430         IOCBQ           *next;
3431         IOCBQ           *prev;
3432         uint32_t        found;
3433         uint32_t        pass = 0;

3435         sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
3436         iocbq = &sbp->iocbq;
3437         nlp = (NODELIST *)sbp->node;
3438         cp = (CHANNEL *)sbp->channel;
3439         channelno = (cp) ? cp->channelno : 0;

3441         if (!(port->flag & EMLXS_INI_BOUND)) {
3442                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3443                     "Port not bound.");
3444                 return (FC_UNBOUND);
3445         }

3447         if (!(hba->flag & FC_ONLINE_MODE)) {
3448                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3449                     "Adapter offline.");
3450                 return (FC_OFFLINE);
3451         }

3453         /* ULP requires the aborted pkt to be completed */
3454         /* back to ULP before returning from this call. */
3455         /* SUN knows of problems with this call so they suggested that we */
3456         /* always return a FC_FAILURE for this call, until it is worked out. */

3458         /* Check if pkt is no good */
3459         if (!(sbp->pkt_flags & PACKET_VALID) ||
3460             (sbp->pkt_flags & PACKET_ULP_OWNED)) {
3461                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3462                     "Bad sbp. flags=%x", sbp->pkt_flags);
3463                 return (FC_FAILURE);
3464         }

3466         /* Tag this now */
3467         /* This will prevent any thread except ours from completing it */
3468         mutex_enter(&sbp->mtx);

3470         /* Check again if we still own this */
3471         if (!(sbp->pkt_flags & PACKET_VALID) ||
3472             (sbp->pkt_flags & PACKET_ULP_OWNED)) {
3473                 mutex_exit(&sbp->mtx);
3474                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3475                     "Bad sbp. flags=%x", sbp->pkt_flags);
3476                 return (FC_FAILURE);
3477         }

3479         /* Check if pkt is a real polled command */
3480         if (!(sbp->pkt_flags & PACKET_IN_ABORT) &&
3481             (sbp->pkt_flags & PACKET_POLLED)) {
```

```
3482                 mutex_exit(&sbp->mtx);

3484                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3485                     "Attempting to abort a polled I/O. sbp=%p flags=%x", sbp,
3486                     sbp->pkt_flags);
3487                 return (FC_FAILURE);
3488         }

3490         sbp->pkt_flags |= PACKET_POLLED;
3491         sbp->pkt_flags |= PACKET_IN_ABORT;

3493         if (sbp->pkt_flags & (PACKET_IN_COMPLETION | PACKET_IN_FLUSH |
3494             PACKET_IN_TIMEOUT)) {
3495                 mutex_exit(&sbp->mtx);

3497                 /* Do nothing, pkt already on its way out */
3498                 goto done;
3499         }

3501         mutex_exit(&sbp->mtx);

3503 begin:
3504         pass++;

3506         mutex_enter(&EMLXS_TX_CHANNEL_LOCK);

3508         if (sbp->pkt_flags & PACKET_IN_TXQ) {
3509                 /* Find it on the queue */
3510                 found = 0;
3511                 if (iocbq->flag & IOCB_PRIORITY) {
3512                         /* Search the priority queue */
3513                         prev = NULL;
3514                         next = (IOCBQ *) nlp->nlp_ptx[channelno].q_first;

3516                         while (next) {
3517                                 if (next == iocbq) {
3518                                         /* Remove it */
3519                                         if (prev) {
3520                                                 prev->next = iocbq->next;
3521                                         }

3523                                         if (nlp->nlp_ptx[channelno].q_last ==
3524                                             (void *)iocbq) {
3525                                                 nlp->nlp_ptx[channelno].q_last =
3526                                                     (void *)prev;
3527                                         }

3529                                         if (nlp->nlp_ptx[channelno].q_first ==
3530                                             (void *)iocbq) {
3531                                                 nlp->nlp_ptx[channelno].
3532                                                     q_first =
3533                                                     (void *)iocbq->next;
3534                                         }

3536                                         nlp->nlp_ptx[channelno].q_cnt--;
3537                                         iocbq->next = NULL;
3538                                         found = 1;
3539                                         break;
3540                                 }

3542                                 prev = next;
3543                                 next = next->next;
3544                         }
3545                 } else {
3546                         /* Search the normal queue */
3547                         prev = NULL;
```



```
3548                         next = (IOCBQ *) nlp->nlp_tx[channelno].q_first;

3550                         while (next) {
3551                                 if (next == iocbq) {
3552                                         /* Remove it */
3553                                         if (prev) {
3554                                                 prev->next = iocbq->next;
3555                                         }

3557                                         if (nlp->nlp_tx[channelno].q_last ==
3558                                             (void *)iocbq) {
3559                                                 nlp->nlp_tx[channelno].q_last =
3560                                                     (void *)prev;
3561                                         }

3563                                         if (nlp->nlp_tx[channelno].q_first ==
3564                                             (void *)iocbq) {
3565                                                 nlp->nlp_tx[channelno].q_first =
3566                                                     (void *)iocbq->next;
3567                                         }

3569                                         nlp->nlp_tx[channelno].q_cnt--;
3570                                         iocbq->next = NULL;
3571                                         found = 1;
3572                                         break;
3573                                 }

3575                                 prev = next;
3576                                 next = (IOCBQ *) next->next;
3577                         }
3578                 }

3580                 if (!found) {
3581                         mutex_exit(&EMLXS_TX_CHANNEL_LOCK);
3582                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_pkt_abort_failed_msg,
3583                             "I/O not found in driver. sbp=%p flags=%x", sbp,
3584                             sbp->pkt_flags);
3585                         goto done;
3586                 }

3588                 /* Check if node still needs servicing */
3589                 if ((nlp->nlp_ptx[channelno].q_first) ||
3590                     (nlp->nlp_tx[channelno].q_first &&
3591                     !(nlp->nlp_flag[channelno] & NLP_CLOSED))) {

3593                         /*
3594                          * If this is the base node,
3595                          * then don't shift the pointers
3596                          */
3597                         /* We want to drain the base node before moving on */
3598                         if (!nlp->nlp_base) {
3599                                 /* Just shift channel queue */
3600                                 /* pointers to next node */
3601                                 cp->nodeq.q_last = (void *) nlp;
3602                                 cp->nodeq.q_first = nlp->nlp_next[channelno];
3603                         }
3604                 } else {
3605                         /* Remove node from channel queue */

3607                         /* If this is the only node on list */
3608                         if (cp->nodeq.q_first == (void *)nlp &&
3609                             cp->nodeq.q_last == (void *)nlp) {
3610                                 cp->nodeq.q_last = NULL;
3611                                 cp->nodeq.q_first = NULL;
3612                                 cp->nodeq.q_cnt = 0;
3613                         } else if (cp->nodeq.q_first == (void *)nlp) {
```

```
3614                                cp->nodeq.q_first = nlp->nlp_next[channelno];
3615                                ((NODELIST *) cp->nodeq.q_last)->
3616                                    nlp_next[channelno] = cp->nodeq.q_first;
3617                                cp->nodeq.q_cnt--;
3618                        } else {
3619                                /*
3620                                 * This is a little more difficult find the
3621                                 * previous node in the circular channel queue
3622                                 */
3623                                prev_nlp = nlp;
3624                                while (prev_nlp->nlp_next[channelno] != nlp) {
3625                                        prev_nlp = prev_nlp->
3626                                            nlp_next[channelno];
3627                                }

3629                                prev_nlp->nlp_next[channelno] =
3630                                    nlp->nlp_next[channelno];

3632                                if (cp->nodeq.q_last == (void *)nlp) {
3633                                        cp->nodeq.q_last = (void *)prev_nlp;
3634                                }
3635                                cp->nodeq.q_cnt--;

3637                        }

3639                        /* Clear node */
3640                        nlp->nlp_next[channelno] = NULL;
3641                }

3643                /* Free the ULPIOTAG and the bmp */
3644                if (hba->sli_mode == EMLXS_HBA_SLI4_MODE) {
3645                        emlxs_sli4_free_xri(port, sbp, sbp->xrip, 1);
3646                } else {
3647                        (void) emlxs_unregister_pkt(cp, sbp->iotag, 1);
3648                }


3651                mutex_exit(&EMLXS_TX_CHANNEL_LOCK);

3653                emlxs_pkt_complete(sbp, IOSTAT_LOCAL_REJECT,
3654                    IOERR_ABORT_REQUESTED, 1);

3656                goto done;
3657        }

3659        mutex_exit(&EMLXS_TX_CHANNEL_LOCK);


3662        /* Check the chip queue */
3663        mutex_enter(&EMLXS_FCTAB_LOCK);

3665        if ((sbp->pkt_flags & PACKET_IN_CHIPQ) &&
3666            !(sbp->pkt_flags & PACKET_XRI_CLOSED) &&
3667            (sbp == hba->fc_table[sbp->iotag])) {

3669                /* Create the abort IOCB */
3670                if (hba->state >= FC_LINK_UP) {
3671                        iocbq =
3672                            emlxs_create_abort_xri_cn(port, sbp->node,
3673                            sbp->iotag, cp, sbp->class, ABORT_TYPE_ABTS);

3675                        mutex_enter(&sbp->mtx);
3676                        sbp->pkt_flags |= PACKET_XRI_CLOSED;
3677                        sbp->ticks =
3678                            hba->timer_tics + (4 * hba->fc_ratov) + 10;
3679                        sbp->abort_attempts++;
```



```
3680                        mutex_exit(&sbp->mtx);
3681                } else {
3682                        iocbq =
3683                            emlxs_create_close_xri_cn(port, sbp->node,
3684                            sbp->iotag, cp);

3686                        mutex_enter(&sbp->mtx);
3687                        sbp->pkt_flags |= PACKET_XRI_CLOSED;
3688                        sbp->ticks = hba->timer_tics + 30;
3689                        sbp->abort_attempts++;
3690                        mutex_exit(&sbp->mtx);
3691                }

3693                mutex_exit(&EMLXS_FCTAB_LOCK);

3695                /* Send this iocbq */
3696                if (iocbq) {
3697                        EMLXS_SLI_ISSUE_IOCB_CMD(hba, cp, iocbq);
3698                        iocbq = NULL;
3699                }

3701                goto done;
3702        }

3704        mutex_exit(&EMLXS_FCTAB_LOCK);

3706        /* Pkt was not on any queues */

3708        /* Check again if we still own this */
3709        if (!(sbp->pkt_flags & PACKET_VALID) ||
3710            (sbp->pkt_flags &
3711            (PACKET_ULP_OWNED | PACKET_IN_COMPLETION |
3712            PACKET_IN_FLUSH | PACKET_IN_TIMEOUT))) {
3713                goto done;
3714        }

3716        if (!sleep) {
3717                return (FC_FAILURE);
3718        }

3720        /* Apparently the pkt was not found.  Let's delay and try again */
3721        if (pass < 5) {
3722                delay(drv_usectohz(5000000));   /* 5 seconds */

3724                /* Check again if we still own this */
3725                if (!(sbp->pkt_flags & PACKET_VALID) ||
3726                    (sbp->pkt_flags &
3727                    (PACKET_ULP_OWNED | PACKET_IN_COMPLETION |
3728                    PACKET_IN_FLUSH | PACKET_IN_TIMEOUT))) {
3729                        goto done;
3730                }

3732                goto begin;
3733        }

3735 force_it:

3737        /* Force the completion now */
3738        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
3739            "Abort: Completing an IO thats not outstanding: %x", sbp->iotag);

3741        /* Now complete it */
3742        emlxs_pkt_complete(sbp, IOSTAT_LOCAL_REJECT, IOERR_ABORT_REQUESTED,
3743            1);

3745 done:
```

```
3747         /* Now wait for the pkt to complete */
3748         if (!(sbp->pkt_flags & PACKET_COMPLETED)) {
3749                 /* Set thread timeout */
3750                 pkt_timeout = emlxs_timeout(hba, 30);

3752                 /* Check for panic situation */
3753                 if (ddi_in_panic()) {

3755                         /*
3756                          * In panic situations there will be one thread with no
3757                          * interrrupts (hard or soft) and no timers
3758                          */

3760                         /*
3761                          * We must manually poll everything in this thread
3762                          * to keep the driver going.
3763                          */

3765                         /* Keep polling the chip until our IO is completed */
3766                         (void) drv_getparm(LBOLT, &time);
3767                         timer = time + drv_sectohz(1);
3767                         timer = time + drv_usectohz(1000000);
3768                         while ((time < pkt_timeout) &&
3769                             !(sbp->pkt_flags & PACKET_COMPLETED)) {
3770                                 EMLXS_SLI_POLL_INTR(hba);
3771                                 (void) drv_getparm(LBOLT, &time);

3773                                 /* Trigger timer checks periodically */
3774                                 if (time >= timer) {
3775                                         emlxs_timer_checks(hba);
3776                                         timer = time + drv_sectohz(1);
3776                                         timer = time + drv_usectohz(1000000);
3777                                 }
3778                         }
3779                 } else {
3780                         /* Wait for IO completion or pkt_timeout */
3781                         mutex_enter(&EMLXS_PKT_LOCK);
3782                         pkt_ret = 0;
3783                         while ((pkt_ret != -1) &&
3784                             !(sbp->pkt_flags & PACKET_COMPLETED)) {
3785                                 pkt_ret =
3786                                     cv_timedwait(&EMLXS_PKT_CV,
3787                                     &EMLXS_PKT_LOCK, pkt_timeout);
3788                         }
3789                         mutex_exit(&EMLXS_PKT_LOCK);
3790                 }

3792                 /* Check if pkt_timeout occured. This is not good. */
3793                 /* Something happened to our IO. */
3794                 if (!(sbp->pkt_flags & PACKET_COMPLETED)) {
3795                         /* Force the completion now */
3796                         goto force_it;
3797                 }
3798         }
3799 #if (EMLXS_MODREVX == EMLXS_MODREV2X)
3800         emlxs_unswap_pkt(sbp);
3801 #endif  /* EMLXS_MODREV2X */

3803         /* Check again if we still own this */
3804         if ((sbp->pkt_flags & PACKET_VALID) &&
3805             !(sbp->pkt_flags & PACKET_ULP_OWNED)) {
3806                 mutex_enter(&sbp->mtx);
3807                 if ((sbp->pkt_flags & PACKET_VALID) &&
3808                     !(sbp->pkt_flags & PACKET_ULP_OWNED)) {
3809                         sbp->pkt_flags |= PACKET_ULP_OWNED;
```



```
3810                 }
3811                 mutex_exit(&sbp->mtx);
3812         }

3814 #ifdef ULP_PATCH5
3815         if (cfg[CFG_ENABLE_PATCH].current & ULP_PATCH5) {
3816                 return (FC_FAILURE);
3817         }
3818 #endif /* ULP_PATCH5 */

3820         return (FC_SUCCESS);

3822 } /* emlxs_fca_pkt_abort() */
______unchanged_portion_omitted_


3865 extern int32_t
3866 emlxs_reset(emlxs_port_t *port, uint32_t cmd)
3867 {
3868         emlxs_hba_t     *hba = HBA;
3869         int             rval;
3870         int             i = 0;
3871         int             ret;
3872         clock_t         timeout;

3874         switch (cmd) {
3875         case FC_FCA_LINK_RESET:

3877                 mutex_enter(&EMLXS_PORT_LOCK);
3878                 if (!(hba->flag & FC_ONLINE_MODE) ||
3879                     (hba->state <= FC_LINK_DOWN)) {
3880                         mutex_exit(&EMLXS_PORT_LOCK);
3881                         return (FC_SUCCESS);
3882                 }

3884                 if (hba->reset_state &
3885                     (FC_LINK_RESET_INP | FC_PORT_RESET_INP)) {
3886                         mutex_exit(&EMLXS_PORT_LOCK);
3887                         return (FC_FAILURE);
3888                 }

3890                 hba->reset_state |= FC_LINK_RESET_INP;
3891                 hba->reset_request |= FC_LINK_RESET;
3892                 mutex_exit(&EMLXS_PORT_LOCK);

3894                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
3895                     "Resetting Link.");

3897                 mutex_enter(&EMLXS_LINKUP_LOCK);
3898                 hba->linkup_wait_flag = TRUE;
3899                 mutex_exit(&EMLXS_LINKUP_LOCK);

3901                 if (emlxs_reset_link(hba, 1, 1)) {
3902                         mutex_enter(&EMLXS_LINKUP_LOCK);
3903                         hba->linkup_wait_flag = FALSE;
3904                         mutex_exit(&EMLXS_LINKUP_LOCK);

3906                         mutex_enter(&EMLXS_PORT_LOCK);
3907                         hba->reset_state &= ~FC_LINK_RESET_INP;
3908                         hba->reset_request &= ~FC_LINK_RESET;
3909                         mutex_exit(&EMLXS_PORT_LOCK);

3911                         return (FC_FAILURE);
3912                 }

3914                 mutex_enter(&EMLXS_LINKUP_LOCK);
```

```
3915                timeout = emlxs_timeout(hba, 60);
3916                ret = 0;
3917                while ((ret != -1) && (hba->linkup_wait_flag == TRUE)) {
3918                        ret =
3919                            cv_timedwait(&EMLXS_LINKUP_CV, &EMLXS_LINKUP_LOCK,
3920                            timeout);
3921                }

3923                hba->linkup_wait_flag = FALSE;
3924                mutex_exit(&EMLXS_LINKUP_LOCK);

3926                mutex_enter(&EMLXS_PORT_LOCK);
3927                hba->reset_state &= ~FC_LINK_RESET_INP;
3928                hba->reset_request &= ~FC_LINK_RESET;
3929                mutex_exit(&EMLXS_PORT_LOCK);

3931                if (ret == -1) {
3932                        return (FC_FAILURE);
3933                }

3935                return (FC_SUCCESS);

3937        case FC_FCA_CORE:
3938 #ifdef DUMP_SUPPORT
3939                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
3940                    "Dumping Core.");

3942                /* Schedule a USER dump */
3943                emlxs_dump(hba, EMLXS_USER_DUMP, 0, 0);

3945                /* Wait for dump to complete */
3946                emlxs_dump_wait(hba);

3948                return (FC_SUCCESS);
3949 #endif /* DUMP_SUPPORT */

3951        case FC_FCA_RESET:
3952        case FC_FCA_RESET_CORE:

3954                mutex_enter(&EMLXS_PORT_LOCK);
3955                if (hba->reset_state & FC_PORT_RESET_INP) {
3956                        mutex_exit(&EMLXS_PORT_LOCK);
3957                        return (FC_FAILURE);
3958                }

3960                hba->reset_state |= FC_PORT_RESET_INP;
3961                hba->reset_request |= (FC_PORT_RESET | FC_LINK_RESET);

3963                /* wait for any pending link resets to complete */
3964                while ((hba->reset_state & FC_LINK_RESET_INP) &&
3965                    (i++ < 1000)) {
3966                        mutex_exit(&EMLXS_PORT_LOCK);
3967                        delay(drv_usectohz(1000));
3968                        mutex_enter(&EMLXS_PORT_LOCK);
3969                }

3971                if (hba->reset_state & FC_LINK_RESET_INP) {
3972                        hba->reset_state &= ~FC_PORT_RESET_INP;
3973                        hba->reset_request &= ~(FC_PORT_RESET | FC_LINK_RESET);
3974                        mutex_exit(&EMLXS_PORT_LOCK);
3975                        return (FC_FAILURE);
3976                }
3977                mutex_exit(&EMLXS_PORT_LOCK);

3979                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
3980                    "Resetting Adapter.");
```



```
3982                rval = FC_SUCCESS;

3984                if (emlxs_offline(hba, 0) == 0) {
3985                        (void) emlxs_online(hba);
3986                } else {
3987                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
3988                            "Adapter reset failed. Device busy.");

3990                        rval = FC_DEVICE_BUSY;
3991                }

3993                mutex_enter(&EMLXS_PORT_LOCK);
3994                hba->reset_state &= ~FC_PORT_RESET_INP;
3995                hba->reset_request &= ~(FC_PORT_RESET | FC_LINK_RESET);
3996                mutex_exit(&EMLXS_PORT_LOCK);

3998                return (rval);

4000        case EMLXS_DFC_RESET_ALL:
4001        case EMLXS_DFC_RESET_ALL_FORCE_DUMP:

4003                mutex_enter(&EMLXS_PORT_LOCK);
4004                if (hba->reset_state & FC_PORT_RESET_INP) {
4005                        mutex_exit(&EMLXS_PORT_LOCK);
4006                        return (FC_FAILURE);
4007                }

4009                hba->reset_state |= FC_PORT_RESET_INP;
4010                hba->reset_request |= (FC_PORT_RESET | FC_LINK_RESET);

4012                /* wait for any pending link resets to complete */
4013                while ((hba->reset_state & FC_LINK_RESET_INP) &&
4014                    (i++ < 1000)) {
4015                        mutex_exit(&EMLXS_PORT_LOCK);
4016                        delay(drv_usectohz(1000));
4017                        mutex_enter(&EMLXS_PORT_LOCK);
4018                }

4020                if (hba->reset_state & FC_LINK_RESET_INP) {
4021                        hba->reset_state &= ~FC_PORT_RESET_INP;
4022                        hba->reset_request &= ~(FC_PORT_RESET | FC_LINK_RESET);
4023                        mutex_exit(&EMLXS_PORT_LOCK);
4024                        return (FC_FAILURE);
4025                }
4026                mutex_exit(&EMLXS_PORT_LOCK);

4028                rval = FC_SUCCESS;

4030                if (cmd == EMLXS_DFC_RESET_ALL) {
4031                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
4032                            "Resetting Adapter (All Firmware Reset).");

4034                        emlxs_sli4_hba_reset_all(hba, 0);
4035                } else {
4036                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
4037                            "Resetting Adapter "
4038                            "(All Firmware Reset, Force Dump).");

4040                        emlxs_sli4_hba_reset_all(hba, 1);
4041                }

4043                mutex_enter(&EMLXS_PORT_LOCK);
4044                hba->reset_state &= ~FC_PORT_RESET_INP;
4045                hba->reset_request &= ~(FC_PORT_RESET | FC_LINK_RESET);
4046                mutex_exit(&EMLXS_PORT_LOCK);
```

```
4048                /* Wait for the timer thread to detect the error condition */
4049                delay(drv_sectohz(1));
4049                delay(drv_usectohz(1000000));

4051                /* Wait for the HBA to re-initialize */
4052                i = 0;
4053                mutex_enter(&EMLXS_PORT_LOCK);
4054                while (!(hba->flag & FC_ONLINE_MODE) && (i++ < 30)) {
4055                        mutex_exit(&EMLXS_PORT_LOCK);
4056                        delay(drv_sectohz(1));
4056                        delay(drv_usectohz(1000000));
4057                        mutex_enter(&EMLXS_PORT_LOCK);
4058                }

4060                if (!(hba->flag & FC_ONLINE_MODE)) {
4061                        rval = FC_FAILURE;
4062                }

4064                mutex_exit(&EMLXS_PORT_LOCK);

4066                return (rval);

4068        default:
4069                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sfs_debug_msg,
4070                    "reset: Unknown command. cmd=%x", cmd);

4072                break;
4073        }

4075        return (FC_FAILURE);

4077 } /* emlxs_reset() */
```

_____unchanged_portion_omitted_

```
**********************************************************
   38636 Wed Aug 19 07:25:01 2015
new/usr/src/uts/common/io/fibre-channel/fca/oce/oce_mbx.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 121 /*
 122  * function to wait till we get a mbox ready after writing to the
 123  * mbox doorbell
 124  *
 125  * dev - software handle to the device
 126  *
 127  * return 0=ready, ETIMEDOUT=>not ready but timed out
 128  */
 129 int
 130 oce_mbox_wait(struct oce_dev *dev, uint32_t tmo_sec)
 131 {
 132         clock_t tmo;
 133         clock_t now, tstamp;
 134         pd_mpu_mbox_db_t mbox_db;

 136         tmo = (tmo_sec > 0) ? drv_sectohz(tmo_sec) :
 136         tmo = (tmo_sec > 0) ? drv_usectohz(tmo_sec * 1000000) :
 137             drv_usectohz(DEFAULT_MQ_MBOX_TIMEOUT);

 139         /* Add the default timeout to wait for a mailbox to complete */
 140         tmo += drv_usectohz(MBX_READY_TIMEOUT);

 142         tstamp = ddi_get_lbolt();
 143         for (;;) {
 144                 now = ddi_get_lbolt();
 145                 if ((now - tstamp) >= tmo) {
 146                         tmo = 0;
 147                         break;
 148                 }

 150                 mbox_db.dw0 = OCE_DB_READ32(dev, PD_MPU_MBOX_DB);
 151                 if (oce_fm_check_acc_handle(dev, dev->db_handle) != DDI_FM_OK) {
 152                         ddi_fm_service_impact(dev->dip, DDI_SERVICE_DEGRADED);
 153                         oce_fm_ereport(dev, DDI_FM_DEVICE_INVAL_STATE);
 154                 }

 156                 if (mbox_db.bits.ready) {
 157                         return (0);
 158                 }
 159                         drv_usecwait(5);
 160         }

 162         return (ETIMEDOUT);
 163 } /* oce_mbox_wait */
______unchanged_portion_omitted_
```

```
**********************************************************
  460409 Wed Aug 19 07:25:01 2015
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_api.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 793 /*
 794  * ql_attach
 795  *      Configure and attach an instance of the driver
 796  *      for a port.
 797  *
 798  * Input:
 799  *      dip = pointer to device information structure.
 800  *      cmd = attach type.
 801  *
 802  * Returns:
 803  *      DDI_SUCCESS or DDI_FAILURE.
 804  *
 805  * Context:
 806  *      Kernel context.
 807  */
 808 static int
 809 ql_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 810 {
 811         off_t                   regsize;
 812         uint32_t                size;
 813         int                     rval, *ptr;
 814         int                     instance;
 815         uint_t                  progress = 0;
 816         char                    *buf;
 817         ushort_t                caps_ptr, cap;
 818         fc_fca_tran_t           *tran;
 819         ql_adapter_state_t      *ha = NULL;

 821         static char *pmcomps[] = {
 822                 NULL,
 823                 PM_LEVEL_D3_STR,                /* Device OFF */
 824                 PM_LEVEL_D0_STR,                /* Device ON */
 825         };

 827         QL_PRINT_3(CE_CONT, "(%d): started, cmd=%xh\n",
 828             ddi_get_instance(dip), cmd);

 830         buf = (char *)(kmem_zalloc(MAXPATHLEN, KM_SLEEP));

 832         switch (cmd) {
 833         case DDI_ATTACH:
 834                 /* first get the instance */
 835                 instance = ddi_get_instance(dip);

 837                 cmn_err(CE_CONT, "!Qlogic %s(%d) FCA Driver v%s\n",
 838                     QL_NAME, instance, QL_VERSION);

 840                 /* Correct OS version? */
 841                 if (ql_os_release_level != 11) {
 842                         cmn_err(CE_WARN, "%s(%d): This driver is for Solaris "
 843                             "11", QL_NAME, instance);
 844                         goto attach_failed;
 845                 }

 847                 /* Hardware is installed in a DMA-capable slot? */
 848                 if (ddi_slaveonly(dip) == DDI_SUCCESS) {
 849                         cmn_err(CE_WARN, "%s(%d): slave only", QL_NAME,
 850                             instance);
 851                         goto attach_failed;
```



```
 852                 }

 854                 /* No support for high-level interrupts */
 855                 if (ddi_intr_hilevel(dip, 0) != 0) {
 856                         cmn_err(CE_WARN, "%s(%d): High level interrupt"
 857                             " not supported", QL_NAME, instance);
 858                         goto attach_failed;
 859                 }

 861                 /* Allocate our per-device-instance structure */
 862                 if (ddi_soft_state_zalloc(ql_state,
 863                     instance) != DDI_SUCCESS) {
 864                         cmn_err(CE_WARN, "%s(%d): soft state alloc failed",
 865                             QL_NAME, instance);
 866                         goto attach_failed;
 867                 }
 868                 progress |= QL_SOFT_STATE_ALLOCED;

 870                 ha = ddi_get_soft_state(ql_state, instance);
 871                 if (ha == NULL) {
 872                         cmn_err(CE_WARN, "%s(%d): can't get soft state",
 873                             QL_NAME, instance);
 874                         goto attach_failed;
 875                 }
 876                 ha->dip = dip;
 877                 ha->instance = instance;
 878                 ha->hba.base_address = ha;
 879                 ha->pha = ha;

 881                 if (ql_el_trace_desc_ctor(ha) != DDI_SUCCESS) {
 882                         cmn_err(CE_WARN, "%s(%d): can't setup el tracing",
 883                             QL_NAME, instance);
 884                         goto attach_failed;
 885                 }

 887                 /* Get extended logging and dump flags. */
 888                 ql_common_properties(ha);

 890                 if (strcmp(ddi_driver_name(ddi_get_parent(dip)),
 891                     "sbus") == 0) {
 892                         EL(ha, "%s SBUS card detected", QL_NAME);
 893                         ha->cfg_flags |= CFG_SBUS_CARD;
 894                 }

 896                 ha->dev = kmem_zalloc(sizeof (*ha->dev) *
 897                     DEVICE_HEAD_LIST_SIZE, KM_SLEEP);

 899                 ha->outstanding_cmds = kmem_zalloc(
 900                     sizeof (*ha->outstanding_cmds) * MAX_OUTSTANDING_COMMANDS,
 901                     KM_SLEEP);

 903                 ha->ub_array = kmem_zalloc(sizeof (*ha->ub_array) *
 904                     QL_UB_LIMIT, KM_SLEEP);

 906                 ha->adapter_stats = kmem_zalloc(sizeof (*ha->adapter_stats),
 907                     KM_SLEEP);

 909                 (void) ddi_pathname(dip, buf);
 910                 ha->devpath = kmem_zalloc(strlen(buf)+1, KM_SLEEP);
 911                 if (ha->devpath == NULL) {
 912                         EL(ha, "devpath mem alloc failed\n");
 913                 } else {
 914                         (void) strcpy(ha->devpath, buf);
 915                         EL(ha, "devpath is: %s\n", ha->devpath);
 916                 }
```

```
                if (CFG_IST(ha, CFG_SBUS_CARD)) {
                        /*
                         * For cards where PCI is mapped to sbus e.g. Ivory.
                         *
                         * 0x00 : 0x000 - 0x0FF PCI Config Space for 2200
                         *      : 0x100 - 0x3FF PCI IO space for 2200
                         * 0x01 : 0x000 - 0x0FF PCI Config Space for fpga
                         *      : 0x100 - 0x3FF PCI IO Space for fpga
                         */
                        if (ddi_regs_map_setup(dip, 0, (caddr_t *)&ha->iobase,
                            0x100, 0x300, &ql_dev_acc_attr, &ha->dev_handle) !=
                            DDI_SUCCESS) {
                                cmn_err(CE_WARN, "%s(%d): Unable to map device"
                                    " registers", QL_NAME, instance);
                                goto attach_failed;
                        }
                        if (ddi_regs_map_setup(dip, 1,
                            (caddr_t *)&ha->sbus_fpga_iobase, 0, 0x400,
                            &ql_dev_acc_attr, &ha->sbus_fpga_dev_handle) !=
                            DDI_SUCCESS) {
                                /* We should not fail attach here */
                                cmn_err(CE_WARN, "%s(%d): Unable to map FPGA",
                                    QL_NAME, instance);
                                ha->sbus_fpga_iobase = NULL;
                        }
                        progress |= QL_REGS_MAPPED;

                        /*
                         * We should map config space before adding interrupt
                         * So that the chip type (2200 or 2300) can be
                         * determined before the interrupt routine gets a
                         * chance to execute.
                         */
                        if (ddi_regs_map_setup(dip, 0,
                            (caddr_t *)&ha->sbus_config_base, 0, 0x100,
                            &ql_dev_acc_attr, &ha->sbus_config_handle) !=
                            DDI_SUCCESS) {
                                cmn_err(CE_WARN, "%s(%d): Unable to map sbus "
                                    "config registers", QL_NAME, instance);
                                goto attach_failed;
                        }
                        progress |= QL_CONFIG_SPACE_SETUP;
                } else {
                        /*LINTED [Solaris DDI_DEV_T_ANY Lint error]*/
                        rval = ddi_prop_lookup_int_array(DDI_DEV_T_ANY, dip,
                            DDI_PROP_DONTPASS, "reg", &ptr, &size);
                        if (rval != DDI_PROP_SUCCESS) {
                                cmn_err(CE_WARN, "%s(%d): Unable to get PCI "
                                    "address registers", QL_NAME, instance);
                                goto attach_failed;
                        } else {
                                ha->pci_bus_addr = ptr[0];
                                ha->function_number = (uint8_t)
                                    (ha->pci_bus_addr >> 8 & 7);
                                ddi_prop_free(ptr);
                        }

                        /*
                         * We should map config space before adding interrupt
                         * So that the chip type (2200 or 2300) can be
                         * determined before the interrupt routine gets a
                         * chance to execute.
                         */
                        if (pci_config_setup(ha->dip, &ha->pci_handle) !=
                            DDI_SUCCESS) {
                                cmn_err(CE_WARN, "%s(%d): can't setup PCI "
```



```
                                    "config space", QL_NAME, instance);
                                goto attach_failed;
                        }
                        progress |= QL_CONFIG_SPACE_SETUP;

                        /*
                         * Setup the ISP2200 registers address mapping to be
                         * accessed by this particular driver.
                         * 0x0   Configuration Space
                         * 0x1   I/O Space
                         * 0x2   32-bit Memory Space address
                         * 0x3   64-bit Memory Space address
                         */
                        size = ql_pci_config_get32(ha, PCI_CONF_BASE0) & BIT_0 ?
                            2 : 1;
                        if (ddi_dev_regsize(dip, size, &regsize) !=
                            DDI_SUCCESS ||
                            ddi_regs_map_setup(dip, size, &ha->iobase,
                            0, regsize, &ql_dev_acc_attr, &ha->dev_handle) !=
                            DDI_SUCCESS) {
                                cmn_err(CE_WARN, "%s(%d): regs_map_setup(mem) "
                                    "failed", QL_NAME, instance);
                                goto attach_failed;
                        }
                        progress |= QL_REGS_MAPPED;

                        /*
                         * We need I/O space mappings for 23xx HBAs for
                         * loading flash (FCode). The chip has a bug due to
                         * which loading flash fails through mem space
                         * mappings in PCI-X mode.
                         */
                        if (size == 1) {
                                ha->iomap_iobase = ha->iobase;
                                ha->iomap_dev_handle = ha->dev_handle;
                        } else {
                                if (ddi_dev_regsize(dip, 1, &regsize) !=
                                    DDI_SUCCESS ||
                                    ddi_regs_map_setup(dip, 1,
                                    &ha->iomap_iobase, 0, regsize,
                                    &ql_dev_acc_attr, &ha->iomap_dev_handle) !=
                                    DDI_SUCCESS) {
                                        cmn_err(CE_WARN, "%s(%d): regs_map_"
                                            "setup(I/O) failed", QL_NAME,
                                            instance);
                                        goto attach_failed;
                                }
                                progress |= QL_IOMAP_IOBASE_MAPPED;
                        }
                }

                ha->subsys_id = (uint16_t)ql_pci_config_get16(ha,
                    PCI_CONF_SUBSYSID);
                ha->subven_id = (uint16_t)ql_pci_config_get16(ha,
                    PCI_CONF_SUBVENID);
                ha->ven_id = (uint16_t)ql_pci_config_get16(ha,
                    PCI_CONF_VENID);
                ha->device_id = (uint16_t)ql_pci_config_get16(ha,
                    PCI_CONF_DEVID);
                ha->rev_id = (uint8_t)ql_pci_config_get8(ha,
                    PCI_CONF_REVID);

                EL(ha, "ISP%x chip detected (RevID=%x, VenID=%x, SVenID=%x, "
                    "SSysID=%x)\n", ha->device_id, ha->rev_id, ha->ven_id,
                    ha->subven_id, ha->subsys_id);
```

```
1050                switch (ha->device_id) {
1051                case 0x2300:
1052                case 0x2312:
1053                case 0x2322:
1054                case 0x6312:
1055                case 0x6322:
1056                        if (ql_pci_config_get8(ha, PCI_CONF_IPIN) == 2) {
1057                                ha->flags |= FUNCTION_1;
1058                        }
1059                        if ((ha->device_id == 0x6322) ||
1060                            (ha->device_id == 0x2322)) {
1061                                ha->cfg_flags |= CFG_CTRL_6322;
1062                                ha->fw_class = 0x6322;
1063                                ha->risc_dump_size = QL_6322_FW_DUMP_SIZE;
1064                        } else {
1065                                ha->cfg_flags |= CFG_CTRL_2300;
1066                                ha->fw_class = 0x2300;
1067                                ha->risc_dump_size = QL_2300_FW_DUMP_SIZE;
1068                        }
1069                        ha->reg_off = &reg_off_2300;
1070                        if (ql_fwmodule_resolve(ha) != QL_SUCCESS) {
1071                                goto attach_failed;
1072                        }
1073                        ha->fcp_cmd = ql_command_iocb;
1074                        ha->ip_cmd = ql_ip_iocb;
1075                        ha->ms_cmd = ql_ms_iocb;
1076                        if (CFG_IST(ha, CFG_SBUS_CARD)) {
1077                                ha->cmd_segs = CMD_TYPE_2_DATA_SEGMENTS;
1078                                ha->cmd_cont_segs = CONT_TYPE_0_DATA_SEGMENTS;
1079                        } else {
1080                                ha->cmd_segs = CMD_TYPE_3_DATA_SEGMENTS;
1081                                ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;
1082                        }
1083                        break;

1085                case 0x2200:
1086                        ha->cfg_flags |= CFG_CTRL_2200;
1087                        ha->reg_off = &reg_off_2200;
1088                        ha->fw_class = 0x2200;
1089                        if (ql_fwmodule_resolve(ha) != QL_SUCCESS) {
1090                                goto attach_failed;
1091                        }
1092                        ha->risc_dump_size = QL_2200_FW_DUMP_SIZE;
1093                        ha->fcp_cmd = ql_command_iocb;
1094                        ha->ip_cmd = ql_ip_iocb;
1095                        ha->ms_cmd = ql_ms_iocb;
1096                        if (CFG_IST(ha, CFG_SBUS_CARD)) {
1097                                ha->cmd_segs = CMD_TYPE_2_DATA_SEGMENTS;
1098                                ha->cmd_cont_segs = CONT_TYPE_0_DATA_SEGMENTS;
1099                        } else {
1100                                ha->cmd_segs = CMD_TYPE_3_DATA_SEGMENTS;
1101                                ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;
1102                        }
1103                        break;

1105                case 0x2422:
1106                case 0x2432:
1107                case 0x5422:
1108                case 0x5432:
1109                case 0x8432:
1110                        if (ql_pci_config_get8(ha, PCI_CONF_IPIN) == 2) {
1111                                ha->flags |= FUNCTION_1;
1112                        }
1113                        ha->cfg_flags |= CFG_CTRL_2422;
1114                        if (ha->device_id == 0x8432) {
1115                                ha->cfg_flags |= CFG_CTRL_MENLO;
```



```
1116                        } else {
1117                                ha->flags |= VP_ENABLED;
1118                        }

1120                        ha->reg_off = &reg_off_2400_2500;
1121                        ha->fw_class = 0x2400;
1122                        if (ql_fwmodule_resolve(ha) != QL_SUCCESS) {
1123                                goto attach_failed;
1124                        }
1125                        ha->risc_dump_size = QL_24XX_FW_DUMP_SIZE;
1126                        ha->fcp_cmd = ql_command_24xx_iocb;
1127                        ha->ip_cmd = ql_ip_24xx_iocb;
1128                        ha->ms_cmd = ql_ms_24xx_iocb;
1129                        ha->els_cmd = ql_els_24xx_iocb;
1130                        ha->cmd_segs = CMD_TYPE_7_DATA_SEGMENTS;
1131                        ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;
1132                        break;

1134                case 0x2522:
1135                case 0x2532:
1136                        if (ql_pci_config_get8(ha, PCI_CONF_IPIN) == 2) {
1137                                ha->flags |= FUNCTION_1;
1138                        }
1139                        ha->cfg_flags |= CFG_CTRL_25XX;
1140                        ha->flags |= VP_ENABLED;
1141                        ha->fw_class = 0x2500;
1142                        ha->reg_off = &reg_off_2400_2500;
1143                        if (ql_fwmodule_resolve(ha) != QL_SUCCESS) {
1144                                goto attach_failed;
1145                        }
1146                        ha->risc_dump_size = QL_25XX_FW_DUMP_SIZE;
1147                        ha->fcp_cmd = ql_command_24xx_iocb;
1148                        ha->ip_cmd = ql_ip_24xx_iocb;
1149                        ha->ms_cmd = ql_ms_24xx_iocb;
1150                        ha->els_cmd = ql_els_24xx_iocb;
1151                        ha->cmd_segs = CMD_TYPE_7_DATA_SEGMENTS;
1152                        ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;
1153                        break;

1155                case 0x8001:
1156                        if (ql_pci_config_get8(ha, PCI_CONF_IPIN) == 4) {
1157                                ha->flags |= FUNCTION_1;
1158                        }
1159                        ha->cfg_flags |= CFG_CTRL_81XX;
1160                        ha->flags |= VP_ENABLED;
1161                        ha->fw_class = 0x8100;
1162                        ha->reg_off = &reg_off_2400_2500;
1163                        if (ql_fwmodule_resolve(ha) != QL_SUCCESS) {
1164                                goto attach_failed;
1165                        }
1166                        ha->risc_dump_size = QL_25XX_FW_DUMP_SIZE;
1167                        ha->fcp_cmd = ql_command_24xx_iocb;
1168                        ha->ip_cmd = ql_ip_24xx_iocb;
1169                        ha->ms_cmd = ql_ms_24xx_iocb;
1170                        ha->cmd_segs = CMD_TYPE_7_DATA_SEGMENTS;
1171                        ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;
1172                        break;

1174                case 0x8021:
1175                        if (ha->function_number & BIT_0) {
1176                                ha->flags |= FUNCTION_1;
1177                        }
1178                        ha->cfg_flags |= CFG_CTRL_8021;
1179                        ha->reg_off = &reg_off_8021;
1180                        ha->risc_dump_size = QL_25XX_FW_DUMP_SIZE;
1181                        ha->fcp_cmd = ql_command_24xx_iocb;
```

```
1182                        ha->ms_cmd = ql_ms_24xx_iocb;
1183                        ha->cmd_segs = CMD_TYPE_7_DATA_SEGMENTS;
1184                        ha->cmd_cont_segs = CONT_TYPE_1_DATA_SEGMENTS;

1186                        ha->nx_pcibase = ha->iobase;
1187                        ha->iobase += 0xBC000 + (ha->function_number << 11);
1188                        ha->iomap_iobase += 0xBC000 +
1189                            (ha->function_number << 11);

1191                        /* map doorbell */
1192                        if (ddi_dev_regsize(dip, 2, &regsize) != DDI_SUCCESS ||
1193                            ddi_regs_map_setup(dip, 2, &ha->db_iobase,
1194                            0, regsize, &ql_dev_acc_attr, &ha->db_dev_handle) !=
1195                            DDI_SUCCESS) {
1196                                cmn_err(CE_WARN, "%s(%d): regs_map_setup"
1197                                    "(doorbell) failed", QL_NAME, instance);
1198                                goto attach_failed;
1199                        }
1200                        progress |= QL_DB_IOBASE_MAPPED;

1202                        ha->nx_req_in = (uint32_t *)(ha->db_iobase +
1203                            (ha->function_number << 12));
1204                        ha->db_read = ha->nx_pcibase + (512 * 1024) +
1205                            (ha->function_number * 8);

1207                        ql_8021_update_crb_int_ptr(ha);
1208                        ql_8021_set_drv_active(ha);
1209                        break;

1211                default:
1212                        cmn_err(CE_WARN, "%s(%d): Unsupported device id: %x",
1213                            QL_NAME, instance, ha->device_id);
1214                        goto attach_failed;
1215                }

1217                /* Setup hba buffer. */

1219                size = CFG_IST(ha, CFG_CTRL_24258081) ?
1220                    (REQUEST_QUEUE_SIZE + RESPONSE_QUEUE_SIZE) :
1221                    (REQUEST_QUEUE_SIZE + RESPONSE_QUEUE_SIZE +
1222                    RCVBUF_QUEUE_SIZE);

1224                if (ql_get_dma_mem(ha, &ha->hba_buf, size, LITTLE_ENDIAN_DMA,
1225                    QL_DMA_RING_ALIGN) != QL_SUCCESS) {
1226                        cmn_err(CE_WARN, "%s(%d): request queue DMA memory "
1227                            "alloc failed", QL_NAME, instance);
1228                        goto attach_failed;
1229                }
1230                progress |= QL_HBA_BUFFER_SETUP;

1232                /* Setup buffer pointers. */
1233                ha->request_dvma = ha->hba_buf.cookie.dmac_laddress +
1234                    REQUEST_Q_BUFFER_OFFSET;
1235                ha->request_ring_bp = (struct cmd_entry *)
1236                    ((caddr_t)ha->hba_buf.bp + REQUEST_Q_BUFFER_OFFSET);

1238                ha->response_dvma = ha->hba_buf.cookie.dmac_laddress +
1239                    RESPONSE_Q_BUFFER_OFFSET;
1240                ha->response_ring_bp = (struct sts_entry *)
1241                    ((caddr_t)ha->hba_buf.bp + RESPONSE_Q_BUFFER_OFFSET);

1243                ha->rcvbuf_dvma = ha->hba_buf.cookie.dmac_laddress +
1244                    RCVBUF_Q_BUFFER_OFFSET;
1245                ha->rcvbuf_ring_bp = (struct rcvbuf *)
1246                    ((caddr_t)ha->hba_buf.bp + RCVBUF_Q_BUFFER_OFFSET);
```



```
1248                /* Allocate resource for QLogic IOCTL */
1249                (void) ql_alloc_xioctl_resource(ha);

1251                /* Setup interrupts */
1252                if ((rval = ql_setup_interrupts(ha)) != DDI_SUCCESS) {
1253                        cmn_err(CE_WARN, "%s(%d): Failed to add interrupt, "
1254                            "rval=%xh", QL_NAME, instance, rval);
1255                        goto attach_failed;
1256                }

1258                progress |= (QL_INTR_ADDED | QL_MUTEX_CV_INITED);

1260                if (ql_nvram_cache_desc_ctor(ha) != DDI_SUCCESS) {
1261                        cmn_err(CE_WARN, "%s(%d): can't setup nvram cache",
1262                            QL_NAME, instance);
1263                        goto attach_failed;
1264                }

1266                /*
1267                 * Allocate an N Port information structure
1268                 * for use when in P2P topology.
1269                 */
1270                ha->n_port = (ql_n_port_info_t *)
1271                    kmem_zalloc(sizeof (ql_n_port_info_t), KM_SLEEP);
1272                if (ha->n_port == NULL) {
1273                        cmn_err(CE_WARN, "%s(%d): Failed to create N Port info",
1274                            QL_NAME, instance);
1275                        goto attach_failed;
1276                }

1278                progress |= QL_N_PORT_INFO_CREATED;

1280                /*
1281                 * Determine support for Power Management
1282                 */
1283                caps_ptr = (uint8_t)ql_pci_config_get8(ha, PCI_CONF_CAP_PTR);

1285                while (caps_ptr != PCI_CAP_NEXT_PTR_NULL) {
1286                        cap = (uint8_t)ql_pci_config_get8(ha, caps_ptr);
1287                        if (cap == PCI_CAP_ID_PM) {
1288                                ha->pm_capable = 1;
1289                                break;
1290                        }
1291                        caps_ptr = (uint8_t)ql_pci_config_get8(ha, caps_ptr +
1292                            PCI_CAP_NEXT_PTR);
1293                }

1295                if (ha->pm_capable) {
1296                        /*
1297                         * Enable PM for 2200 based HBAs only.
1298                         */
1299                        if (ha->device_id != 0x2200) {
1300                                ha->pm_capable = 0;
1301                        }
1302                }

1304                if (ha->pm_capable) {
1305                        ha->pm_capable = ql_enable_pm;
1306                }

1308                if (ha->pm_capable) {
1309                        /*
1310                         * Initialize power management bookkeeping;
1311                         * components are created idle.
1312                         */
1313                        (void) sprintf(buf, "NAME=%s(%d)", QL_NAME, instance);
```

```
1314                        pmcomps[0] = buf;
1316                        /*LINTED [Solaris DDI_DEV_T_NONE Lint warning]*/
1317                        if (ddi_prop_update_string_array(DDI_DEV_T_NONE,
1318                            dip, "pm-components", pmcomps,
1319                            sizeof (pmcomps) / sizeof (pmcomps[0])) !=
1320                            DDI_PROP_SUCCESS) {
1321                                cmn_err(CE_WARN, "%s(%d): failed to create"
1322                                    " pm-components property", QL_NAME,
1323                                    instance);
1325                                /* Initialize adapter. */
1326                                ha->power_level = PM_LEVEL_D0;
1327                                if (ql_initialize_adapter(ha) != QL_SUCCESS) {
1328                                        cmn_err(CE_WARN, "%s(%d): failed to"
1329                                            " initialize adapter", QL_NAME,
1330                                            instance);
1331                                        goto attach_failed;
1332                                }
1333                        } else {
1334                                ha->power_level = PM_LEVEL_D3;
1335                                if (pm_raise_power(dip, QL_POWER_COMPONENT,
1336                                    PM_LEVEL_D0) != DDI_SUCCESS) {
1337                                        cmn_err(CE_WARN, "%s(%d): failed to"
1338                                            " raise power or initialize"
1339                                            " adapter", QL_NAME, instance);
1340                                }
1341                        }
1342                } else {
1343                        /* Initialize adapter. */
1344                        ha->power_level = PM_LEVEL_D0;
1345                        if (ql_initialize_adapter(ha) != QL_SUCCESS) {
1346                                cmn_err(CE_WARN, "%s(%d): failed to initialize"
1347                                    " adapter", QL_NAME, instance);
1348                        }
1349                }
1351                if (ha->fw_major_version == 0 && ha->fw_minor_version == 0 &&
1352                    ha->fw_subminor_version == 0) {
1353                        cmn_err(CE_NOTE, "!%s(%d): Firmware not loaded",
1354                            QL_NAME, ha->instance);
1355                } else {
1356                        int     rval;
1357                        char    ver_fmt[256];
1359                        rval = (int)snprintf(ver_fmt, (size_t)sizeof (ver_fmt),
1360                            "Firmware version %d.%d.%d", ha->fw_major_version,
1361                            ha->fw_minor_version, ha->fw_subminor_version);
1363                        if (CFG_IST(ha, CFG_CTRL_81XX)) {
1364                                rval = (int)snprintf(ver_fmt + rval,
1365                                    (size_t)sizeof (ver_fmt),
1366                                    ", MPI fw version %d.%d.%d",
1367                                    ha->mpi_fw_major_version,
1368                                    ha->mpi_fw_minor_version,
1369                                    ha->mpi_fw_subminor_version);
1371                                if (ha->subsys_id == 0x17B ||
1372                                    ha->subsys_id == 0x17D) {
1373                                        (void) snprintf(ver_fmt + rval,
1374                                            (size_t)sizeof (ver_fmt),
1375                                            ", PHY fw version %d.%d.%d",
1376                                            ha->phy_fw_major_version,
1377                                            ha->phy_fw_minor_version,
1378                                            ha->phy_fw_subminor_version);
1379                                }
```



```
1380                        }
1381                        cmn_err(CE_NOTE, "!%s(%d): %s",
1382                            QL_NAME, ha->instance, ver_fmt);
1383                }
1385                ha->k_stats = kstat_create(QL_NAME, instance, "statistics",
1386                    "controller", KSTAT_TYPE_RAW,
1387                    (uint32_t)sizeof (ql_adapter_stat_t), KSTAT_FLAG_VIRTUAL);
1388                if (ha->k_stats == NULL) {
1389                        cmn_err(CE_WARN, "%s(%d): Failed to create kstat",
1390                            QL_NAME, instance);
1391                        goto attach_failed;
1392                }
1393                progress |= QL_KSTAT_CREATED;
1395                ha->adapter_stats->version = 1;
1396                ha->k_stats->ks_data = (void *)ha->adapter_stats;
1397                ha->k_stats->ks_private = ha;
1398                ha->k_stats->ks_update = ql_kstat_update;
1399                ha->k_stats->ks_ndata = 1;
1400                ha->k_stats->ks_data_size = sizeof (ql_adapter_stat_t);
1401                kstat_install(ha->k_stats);
1403                if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
1404                    instance, DDI_NT_NEXUS, 0) != DDI_SUCCESS) {
1405                        cmn_err(CE_WARN, "%s(%d): failed to create minor node",
1406                            QL_NAME, instance);
1407                        goto attach_failed;
1408                }
1409                progress |= QL_MINOR_NODE_CREATED;
1411                /* Allocate a transport structure for this instance */
1412                tran = kmem_zalloc(sizeof (fc_fca_tran_t), KM_SLEEP);
1413                if (tran == NULL) {
1414                        cmn_err(CE_WARN, "%s(%d): failed to allocate transport",
1415                            QL_NAME, instance);
1416                        goto attach_failed;
1417                }
1419                progress |= QL_FCA_TRAN_ALLOCED;
1421                /* fill in the structure */
1422                tran->fca_numports = 1;
1423                tran->fca_version = FCTL_FCA_MODREV_5;
1424                if (CFG_IST(ha, CFG_CTRL_2422)) {
1425                        tran->fca_num_npivports = MAX_24_VIRTUAL_PORTS;
1426                } else if (CFG_IST(ha, CFG_CTRL_2581)) {
1427                        tran->fca_num_npivports = MAX_25_VIRTUAL_PORTS;
1428                }
1429                bcopy(ha->loginparams.node_ww_name.raw_wwn,
1430                    tran->fca_perm_pwwn.raw_wwn, 8);
1432                EL(ha, "FCA version %d\n", tran->fca_version);
1434                /* Specify the amount of space needed in each packet */
1435                tran->fca_pkt_size = sizeof (ql_srb_t);
1437                /* command limits are usually dictated by hardware */
1438                tran->fca_cmd_max = MAX_OUTSTANDING_COMMANDS;
1440                /* dmaattr are static, set elsewhere. */
1441                if (CFG_IST(ha, CFG_ENABLE_64BIT_ADDRESSING)) {
1442                        tran->fca_dma_attr = &ql_64bit_io_dma_attr;
1443                        tran->fca_dma_fcp_cmd_attr = &ql_64fcp_cmd_dma_attr;
1444                        tran->fca_dma_fcp_rsp_attr = &ql_64fcp_rsp_dma_attr;
1445                        tran->fca_dma_fcp_data_attr = &ql_64fcp_data_dma_attr;
```

```
1446                        tran->fca_dma_fcsm_cmd_attr = &ql_64fcsm_cmd_dma_attr;
1447                        tran->fca_dma_fcsm_rsp_attr = &ql_64fcsm_rsp_dma_attr;
1448                        tran->fca_dma_fcip_cmd_attr = &ql_64fcip_cmd_dma_attr;
1449                        tran->fca_dma_fcip_rsp_attr = &ql_64fcip_rsp_dma_attr;
1450                } else {
1451                        tran->fca_dma_attr = &ql_32bit_io_dma_attr;
1452                        tran->fca_dma_fcp_cmd_attr = &ql_32fcp_cmd_dma_attr;
1453                        tran->fca_dma_fcp_rsp_attr = &ql_32fcp_rsp_dma_attr;
1454                        tran->fca_dma_fcp_data_attr = &ql_32fcp_data_dma_attr;
1455                        tran->fca_dma_fcsm_cmd_attr = &ql_32fcsm_cmd_dma_attr;
1456                        tran->fca_dma_fcsm_rsp_attr = &ql_32fcsm_rsp_dma_attr;
1457                        tran->fca_dma_fcip_cmd_attr = &ql_32fcip_cmd_dma_attr;
1458                        tran->fca_dma_fcip_rsp_attr = &ql_32fcip_rsp_dma_attr;
1459                }

1461                tran->fca_acc_attr = &ql_dev_acc_attr;
1462                tran->fca_iblock = &(ha->iblock_cookie);

1464                /* the remaining values are simply function vectors */
1465                tran->fca_bind_port = ql_bind_port;
1466                tran->fca_unbind_port = ql_unbind_port;
1467                tran->fca_init_pkt = ql_init_pkt;
1468                tran->fca_un_init_pkt = ql_un_init_pkt;
1469                tran->fca_els_send = ql_els_send;
1470                tran->fca_get_cap = ql_get_cap;
1471                tran->fca_set_cap = ql_set_cap;
1472                tran->fca_getmap = ql_getmap;
1473                tran->fca_transport = ql_transport;
1474                tran->fca_ub_alloc = ql_ub_alloc;
1475                tran->fca_ub_free = ql_ub_free;
1476                tran->fca_ub_release = ql_ub_release;
1477                tran->fca_abort = ql_abort;
1478                tran->fca_reset = ql_reset;
1479                tran->fca_port_manage = ql_port_manage;
1480                tran->fca_get_device = ql_get_device;

1482                /* give it to the FC transport */
1483                if (fc_fca_attach(dip, tran) != DDI_SUCCESS) {
1484                        cmn_err(CE_WARN, "%s(%d): FCA attach failed", QL_NAME,
1485                            instance);
1486                        goto attach_failed;
1487                }
1488                progress |= QL_FCA_ATTACH_DONE;

1490                /* Stash the structure so it can be freed at detach */
1491                ha->tran = tran;

1493                /* Acquire global state lock. */
1494                GLOBAL_STATE_LOCK();

1496                /* Add adapter structure to link list. */
1497                ql_add_link_b(&ql_hba, &ha->hba);

1499                /* Start one second driver timer. */
1500                if (ql_timer_timeout_id == NULL) {
1501                        ql_timer_ticks = drv_sectohz(1);
1501                        ql_timer_ticks = drv_usectohz(1000000);
1502                        ql_timer_timeout_id = timeout(ql_timer, (void *)0,
1503                            ql_timer_ticks);
1504                }

1506                /* Release global state lock. */
1507                GLOBAL_STATE_UNLOCK();

1509                /* Determine and populate HBA fru info */
1510                ql_setup_fruinfo(ha);
```



```
1512                /* Setup task_daemon thread. */
1513                (void) thread_create(NULL, 0, (void (*)())ql_task_daemon, ha,
1514                    0, &p0, TS_RUN, minclsyspri);

1516                progress |= QL_TASK_DAEMON_STARTED;

1518                ddi_report_dev(dip);

1520                /* Disable link reset in panic path */
1521                ha->lip_on_panic = 1;

1523                rval = DDI_SUCCESS;
1524                break;

1526 attach_failed:
1527                if (progress & QL_FCA_ATTACH_DONE) {
1528                        (void) fc_fca_detach(dip);
1529                        progress &= ~QL_FCA_ATTACH_DONE;
1530                }

1532                if (progress & QL_FCA_TRAN_ALLOCED) {
1533                        kmem_free(tran, sizeof (fc_fca_tran_t));
1534                        progress &= ~QL_FCA_TRAN_ALLOCED;
1535                }

1537                if (progress & QL_MINOR_NODE_CREATED) {
1538                        ddi_remove_minor_node(dip, "devctl");
1539                        progress &= ~QL_MINOR_NODE_CREATED;
1540                }

1542                if (progress & QL_KSTAT_CREATED) {
1543                        kstat_delete(ha->k_stats);
1544                        progress &= ~QL_KSTAT_CREATED;
1545                }

1547                if (progress & QL_N_PORT_INFO_CREATED) {
1548                        kmem_free(ha->n_port, sizeof (ql_n_port_info_t));
1549                        progress &= ~QL_N_PORT_INFO_CREATED;
1550                }

1552                if (progress & QL_TASK_DAEMON_STARTED) {
1553                        TASK_DAEMON_LOCK(ha);

1555                        ha->task_daemon_flags |= TASK_DAEMON_STOP_FLG;

1557                        cv_signal(&ha->cv_task_daemon);

1559                        /* Release task daemon lock. */
1560                        TASK_DAEMON_UNLOCK(ha);

1562                        /* Wait for for task daemon to stop running. */
1563                        while (ha->task_daemon_flags & TASK_DAEMON_STOP_FLG) {
1564                                ql_delay(ha, 10000);
1565                        }
1566                        progress &= ~QL_TASK_DAEMON_STARTED;
1567                }

1569                if (progress & QL_DB_IOBASE_MAPPED) {
1570                        ql_8021_clr_drv_active(ha);
1571                        ddi_regs_map_free(&ha->db_dev_handle);
1572                        progress &= ~QL_DB_IOBASE_MAPPED;
1573                }
1574                if (progress & QL_IOMAP_IOBASE_MAPPED) {
1575                        ddi_regs_map_free(&ha->iomap_dev_handle);
1576                        progress &= ~QL_IOMAP_IOBASE_MAPPED;
```

```
1577                }

1579                if (progress & QL_CONFIG_SPACE_SETUP) {
1580                        if (CFG_IST(ha, CFG_SBUS_CARD)) {
1581                                ddi_regs_map_free(&ha->sbus_config_handle);
1582                        } else {
1583                                pci_config_teardown(&ha->pci_handle);
1584                        }
1585                        progress &= ~QL_CONFIG_SPACE_SETUP;
1586                }

1588                if (progress & QL_INTR_ADDED) {
1589                        ql_disable_intr(ha);
1590                        ql_release_intr(ha);
1591                        progress &= ~QL_INTR_ADDED;
1592                }

1594                if (progress & QL_MUTEX_CV_INITED) {
1595                        ql_destroy_mutex(ha);
1596                        progress &= ~QL_MUTEX_CV_INITED;
1597                }

1599                if (progress & QL_HBA_BUFFER_SETUP) {
1600                        ql_free_phys(ha, &ha->hba_buf);
1601                        progress &= ~QL_HBA_BUFFER_SETUP;
1602                }

1604                if (progress & QL_REGS_MAPPED) {
1605                        ddi_regs_map_free(&ha->dev_handle);
1606                        if (ha->sbus_fpga_iobase != NULL) {
1607                                ddi_regs_map_free(&ha->sbus_fpga_dev_handle);
1608                        }
1609                        progress &= ~QL_REGS_MAPPED;
1610                }

1612                if (progress & QL_SOFT_STATE_ALLOCED) {

1614                        ql_fcache_rel(ha->fcache);

1616                        kmem_free(ha->adapter_stats,
1617                            sizeof (*ha->adapter_stats));

1619                        kmem_free(ha->ub_array, sizeof (*ha->ub_array) *
1620                            QL_UB_LIMIT);

1622                        kmem_free(ha->outstanding_cmds,
1623                            sizeof (*ha->outstanding_cmds) *
1624                            MAX_OUTSTANDING_COMMANDS);

1626                        if (ha->devpath != NULL) {
1627                                kmem_free(ha->devpath,
1628                                    strlen(ha->devpath) + 1);
1629                        }

1631                        kmem_free(ha->dev, sizeof (*ha->dev) *
1632                            DEVICE_HEAD_LIST_SIZE);

1634                        if (ha->xioctl != NULL) {
1635                                ql_free_xioctl_resource(ha);
1636                        }

1638                        if (ha->fw_module != NULL) {
1639                                (void) ddi_modclose(ha->fw_module);
1640                        }
1641                        (void) ql_el_trace_desc_dtor(ha);
1642                        (void) ql_nvram_cache_desc_dtor(ha);
```



```
1644                        ddi_soft_state_free(ql_state, instance);
1645                        progress &= ~QL_SOFT_STATE_ALLOCED;
1646                }

1648                ddi_prop_remove_all(dip);
1649                rval = DDI_FAILURE;
1650                break;

1652        case DDI_RESUME:
1653                rval = DDI_FAILURE;

1655                ha = ddi_get_soft_state(ql_state, ddi_get_instance(dip));
1656                if (ha == NULL) {
1657                        cmn_err(CE_WARN, "%s(%d): can't get soft state",
1658                            QL_NAME, instance);
1659                        break;
1660                }

1662                ha->power_level = PM_LEVEL_D3;
1663                if (ha->pm_capable) {
1664                        /*
1665                         * Get ql_power to do power on initialization
1666                         */
1667                        if (pm_raise_power(dip, QL_POWER_COMPONENT,
1668                            PM_LEVEL_D0) != DDI_SUCCESS) {
1669                                cmn_err(CE_WARN, "%s(%d): can't raise adapter"
1670                                    " power", QL_NAME, instance);
1671                        }
1672                }

1674                /*
1675                 * There is a bug in DR that prevents PM framework
1676                 * from calling ql_power.
1677                 */
1678                if (ha->power_level == PM_LEVEL_D3) {
1679                        ha->power_level = PM_LEVEL_D0;

1681                        if (ql_initialize_adapter(ha) != QL_SUCCESS) {
1682                                cmn_err(CE_WARN, "%s(%d): can't initialize the"
1683                                    " adapter", QL_NAME, instance);
1684                        }

1686                        /* Wake up task_daemon. */
1687                        ql_awaken_task_daemon(ha, NULL, TASK_DAEMON_ALIVE_FLG,
1688                            0);
1689                }

1691                /* Acquire global state lock. */
1692                GLOBAL_STATE_LOCK();

1694                /* Restart driver timer. */
1695                if (ql_timer_timeout_id == NULL) {
1696                        ql_timer_timeout_id = timeout(ql_timer, (void *)0,
1697                            ql_timer_ticks);
1698                }

1700                /* Release global state lock. */
1701                GLOBAL_STATE_UNLOCK();

1703                /* Wake up command start routine. */
1704                ADAPTER_STATE_LOCK(ha);
1705                ha->flags &= ~ADAPTER_SUSPENDED;
1706                ADAPTER_STATE_UNLOCK(ha);

1708                /*
```

```
1709                     * Transport doesn't make FC discovery in polled
1710                     * mode; So we need the daemon thread's services
1711                     * right here.
1712                     */
1713                    (void) callb_generic_cpr(&ha->cprinfo, CB_CODE_CPR_RESUME);

1715                    rval = DDI_SUCCESS;

1717                    /* Restart IP if it was running. */
1718                    if (ha->flags & IP_ENABLED && !(ha->flags & IP_INITIALIZED)) {
1719                            (void) ql_initialize_ip(ha);
1720                            ql_isp_rcvbuf(ha);
1721                    }
1722                    break;

1724            default:
1725                    cmn_err(CE_WARN, "%s(%d): attach, unknown code:"
1726                        " %x", QL_NAME, ddi_get_instance(dip), cmd);
1727                    rval = DDI_FAILURE;
1728                    break;
1729            }

1731            kmem_free(buf, MAXPATHLEN);

1733            if (rval != DDI_SUCCESS) {
1734                    /*EMPTY*/
1735                    QL_PRINT_2(CE_CONT, "(%d): failed, rval = %xh\n",
1736                        ddi_get_instance(dip), rval);
1737            } else {
1738                    /*EMPTY*/
1739                    QL_PRINT_3(CE_CONT, "(%d): done\n", ddi_get_instance(dip));
1740            }

1742            return (rval);
1743    }

1745    /*
1746     * ql_detach
1747     *      Used to remove all the states associated with a given
1748     *      instances of a device node prior to the removal of that
1749     *      instance from the system.
1750     *
1751     * Input:
1752     *      dip = pointer to device information structure.
1753     *      cmd = type of detach.
1754     *
1755     * Returns:
1756     *      DDI_SUCCESS or DDI_FAILURE.
1757     *
1758     * Context:
1759     *      Kernel context.
1760     */
1761    static int
1762    ql_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
1763    {
1764            ql_adapter_state_t      *ha, *vha;
1765            ql_tgt_t                *tq;
1766            int                     delay_cnt;
1767            uint16_t                index;
1768            ql_link_t               *link;
1769            char                    *buf;
1770            timeout_id_t            timer_id = NULL;
1771            int                     suspend, rval = DDI_SUCCESS;

1773            ha = ddi_get_soft_state(ql_state, ddi_get_instance(dip));
1774            if (ha == NULL) {
```



```
1775                    QL_PRINT_2(CE_CONT, "(%d): no adapter\n",
1776                        ddi_get_instance(dip));
1777                    return (DDI_FAILURE);
1778            }

1780            QL_PRINT_3(CE_CONT, "(%d): started, cmd=%xh\n", ha->instance, cmd);

1782            buf = (char *)(kmem_zalloc(MAXPATHLEN, KM_SLEEP));

1784            switch (cmd) {
1785            case DDI_DETACH:
1786                    ADAPTER_STATE_LOCK(ha);
1787                    ha->flags |= (ADAPTER_SUSPENDED | ABORT_CMDS_LOOP_DOWN_TMO);
1788                    ADAPTER_STATE_UNLOCK(ha);

1790                    TASK_DAEMON_LOCK(ha);

1792                    if (ha->task_daemon_flags & TASK_DAEMON_ALIVE_FLG) {
1793                            ha->task_daemon_flags |= TASK_DAEMON_STOP_FLG;
1794                            cv_signal(&ha->cv_task_daemon);

1796                            TASK_DAEMON_UNLOCK(ha);

1798                            (void) ql_wait_for_td_stop(ha);

1800                            TASK_DAEMON_LOCK(ha);
1801                            if (ha->task_daemon_flags & TASK_DAEMON_STOP_FLG) {
1802                                    ha->task_daemon_flags &= ~TASK_DAEMON_STOP_FLG;
1803                                    EL(ha, "failed, could not stop task daemon\n");
1804                            }
1805                    }
1806                    TASK_DAEMON_UNLOCK(ha);

1808                    GLOBAL_STATE_LOCK();

1810                    /* Disable driver timer if no adapters. */
1811                    if (ql_timer_timeout_id && ql_hba.first == &ha->hba &&
1812                        ql_hba.last == &ha->hba) {
1813                            timer_id = ql_timer_timeout_id;
1814                            ql_timer_timeout_id = NULL;
1815                    }
1816                    ql_remove_link(&ql_hba, &ha->hba);

1818                    GLOBAL_STATE_UNLOCK();

1820                    if (timer_id) {
1821                            (void) untimeout(timer_id);
1822                    }

1824                    if (ha->pm_capable) {
1825                            if (pm_lower_power(dip, QL_POWER_COMPONENT,
1826                                PM_LEVEL_D3) != DDI_SUCCESS) {
1827                                    cmn_err(CE_WARN, "%s(%d): failed to lower the"
1828                                        " power", QL_NAME, ha->instance);
1829                            }
1830                    }

1832                    /*
1833                     * If pm_lower_power shutdown the adapter, there
1834                     * isn't much else to do
1835                     */
1836                    if (ha->power_level != PM_LEVEL_D3) {
1837                            ql_halt(ha, PM_LEVEL_D3);
1838                    }

1840                    /* Remove virtual ports. */
```

```
1841                while ((vha = ha->vp_next) != NULL) {
1842                        ql_vport_destroy(vha);
1843                }

1845                /* Free target queues. */
1846                for (index = 0; index < DEVICE_HEAD_LIST_SIZE; index++) {
1847                        link = ha->dev[index].first;
1848                        while (link != NULL) {
1849                                tq = link->base_address;
1850                                link = link->next;
1851                                ql_dev_free(ha, tq);
1852                        }
1853                }

1855                /*
1856                 * Free unsolicited buffers.
1857                 * If we are here then there are no ULPs still
1858                 * alive that wish to talk to ql so free up
1859                 * any SRB_IP_UB_UNUSED buffers that are
1860                 * lingering around
1861                 */
1862                QL_UB_LOCK(ha);
1863                for (index = 0; index < QL_UB_LIMIT; index++) {
1864                        fc_unsol_buf_t *ubp = ha->ub_array[index];

1866                        if (ubp != NULL) {
1867                                ql_srb_t *sp = ubp->ub_fca_private;

1869                                sp->flags |= SRB_UB_FREE_REQUESTED;

1871                                while (!(sp->flags & SRB_UB_IN_FCA) ||
1872                                    (sp->flags & (SRB_UB_CALLBACK |
1873                                    SRB_UB_ACQUIRED))) {
1874                                        QL_UB_UNLOCK(ha);
1875                                        delay(drv_usectohz(100000));
1876                                        QL_UB_LOCK(ha);
1877                                }
1878                                ha->ub_array[index] = NULL;

1880                                QL_UB_UNLOCK(ha);
1881                                ql_free_unsolicited_buffer(ha, ubp);
1882                                QL_UB_LOCK(ha);
1883                        }
1884                }
1885                QL_UB_UNLOCK(ha);

1887                /* Free any saved RISC code. */
1888                if (ha->risc_code != NULL) {
1889                        kmem_free(ha->risc_code, ha->risc_code_size);
1890                        ha->risc_code = NULL;
1891                        ha->risc_code_size = 0;
1892                }

1894                if (ha->fw_module != NULL) {
1895                        (void) ddi_modclose(ha->fw_module);
1896                        ha->fw_module = NULL;
1897                }

1899                /* Free resources. */
1900                ddi_prop_remove_all(dip);
1901                (void) fc_fca_detach(dip);
1902                kmem_free(ha->tran, sizeof (fc_fca_tran_t));
1903                ddi_remove_minor_node(dip, "devctl");
1904                if (ha->k_stats != NULL) {
1905                        kstat_delete(ha->k_stats);
1906                }
```



```
1908                if (CFG_IST(ha, CFG_SBUS_CARD)) {
1909                        ddi_regs_map_free(&ha->sbus_config_handle);
1910                } else {
1911                        if (CFG_IST(ha, CFG_CTRL_8021)) {
1912                                ql_8021_clr_drv_active(ha);
1913                                ddi_regs_map_free(&ha->db_dev_handle);
1914                        }
1915                        if (ha->iomap_dev_handle != ha->dev_handle) {
1916                                ddi_regs_map_free(&ha->iomap_dev_handle);
1917                        }
1918                        pci_config_teardown(&ha->pci_handle);
1919                }

1921                ql_disable_intr(ha);
1922                ql_release_intr(ha);

1924                ql_free_xioctl_resource(ha);

1926                ql_destroy_mutex(ha);

1928                ql_free_phys(ha, &ha->hba_buf);
1929                ql_free_phys(ha, &ha->fwexttracebuf);
1930                ql_free_phys(ha, &ha->fwfcetracebuf);

1932                ddi_regs_map_free(&ha->dev_handle);
1933                if (ha->sbus_fpga_iobase != NULL) {
1934                        ddi_regs_map_free(&ha->sbus_fpga_dev_handle);
1935                }

1937                ql_fcache_rel(ha->fcache);
1938                if (ha->vcache != NULL) {
1939                        kmem_free(ha->vcache, QL_24XX_VPD_SIZE);
1940                }

1942                if (ha->pi_attrs != NULL) {
1943                        kmem_free(ha->pi_attrs, sizeof (fca_port_attrs_t));
1944                }

1946                kmem_free(ha->adapter_stats, sizeof (*ha->adapter_stats));

1948                kmem_free(ha->ub_array, sizeof (*ha->ub_array) * QL_UB_LIMIT);

1950                kmem_free(ha->outstanding_cmds,
1951                    sizeof (*ha->outstanding_cmds) * MAX_OUTSTANDING_COMMANDS);

1953                if (ha->n_port != NULL) {
1954                        kmem_free(ha->n_port, sizeof (ql_n_port_info_t));
1955                }

1957                if (ha->devpath != NULL) {
1958                        kmem_free(ha->devpath, strlen(ha->devpath) + 1);
1959                }

1961                kmem_free(ha->dev, sizeof (*ha->dev) * DEVICE_HEAD_LIST_SIZE);

1963                EL(ha, "detached\n");

1965                ddi_soft_state_free(ql_state, (int)ha->instance);

1967                break;

1969        case DDI_SUSPEND:
1970                ADAPTER_STATE_LOCK(ha);

1972                delay_cnt = 0;
```

```
1973                ha->flags |= ADAPTER_SUSPENDED;
1974                while (ha->flags & ADAPTER_TIMER_BUSY && delay_cnt++ < 10) {
1975                        ADAPTER_STATE_UNLOCK(ha);
1976                        delay(drv_sectohz(1));
1976                        delay(drv_usectohz(1000000));
1977                        ADAPTER_STATE_LOCK(ha);
1978                }
1979                if (ha->busy || ha->flags & ADAPTER_TIMER_BUSY) {
1980                        ha->flags &= ~ADAPTER_SUSPENDED;
1981                        ADAPTER_STATE_UNLOCK(ha);
1982                        rval = DDI_FAILURE;
1983                        cmn_err(CE_WARN, "!%s(%d): Fail suspend"
1984                            " busy %xh flags %xh", QL_NAME, ha->instance,
1985                            ha->busy, ha->flags);
1986                        break;
1987                }

1989                ADAPTER_STATE_UNLOCK(ha);

1991                if (ha->flags & IP_INITIALIZED) {
1992                        (void) ql_shutdown_ip(ha);
1993                }

1995                if ((suspend = ql_suspend_adapter(ha)) != QL_SUCCESS) {
1996                        ADAPTER_STATE_LOCK(ha);
1997                        ha->flags &= ~ADAPTER_SUSPENDED;
1998                        ADAPTER_STATE_UNLOCK(ha);
1999                        cmn_err(CE_WARN, "%s(%d): Fail suspend rval %xh",
2000                            QL_NAME, ha->instance, suspend);

2002                        /* Restart IP if it was running. */
2003                        if (ha->flags & IP_ENABLED &&
2004                            !(ha->flags & IP_INITIALIZED)) {
2005                                (void) ql_initialize_ip(ha);
2006                                ql_isp_rcvbuf(ha);
2007                        }
2008                        rval = DDI_FAILURE;
2009                        break;
2010                }

2012                /* Acquire global state lock. */
2013                GLOBAL_STATE_LOCK();

2015                /* Disable driver timer if last adapter. */
2016                if (ql_timer_timeout_id && ql_hba.first == &ha->hba &&
2017                    ql_hba.last == &ha->hba) {
2018                        timer_id = ql_timer_timeout_id;
2019                        ql_timer_timeout_id = NULL;
2020                }
2021                GLOBAL_STATE_UNLOCK();

2023                if (timer_id) {
2024                        (void) untimeout(timer_id);
2025                }

2027                EL(ha, "suspended\n");

2029                break;

2031        default:
2032                rval = DDI_FAILURE;
2033                break;
2034        }

2036        kmem_free(buf, MAXPATHLEN);
```



```
2038        if (rval != DDI_SUCCESS) {
2039                if (ha != NULL) {
2040                        EL(ha, "failed, rval = %xh\n", rval);
2041                } else {
2042                        /*EMPTY*/
2043                        QL_PRINT_2(CE_CONT, "(%d): failed, rval = %xh\n",
2044                            ddi_get_instance(dip), rval);
2045                }
2046        } else {
2047                /*EMPTY*/
2048                QL_PRINT_3(CE_CONT, "(%d): done\n", ddi_get_instance(dip));
2049        }

2051        return (rval);
2052 }
______unchanged_portion_omitted_

11813 /*
11814  * ql_binary_fw_dump
11815  *      Dumps binary data from firmware.
11816  *
11817  * Input:
11818  *      ha = adapter state pointer.
11819  *      lock_needed = mailbox lock needed.
11820  *
11821  * Returns:
11822  *      ql local function return status code.
11823  *
11824  * Context:
11825  *      Interrupt or Kernel context, no mailbox commands allowed.
11826  */
11827 int
11828 ql_binary_fw_dump(ql_adapter_state_t *vha, int lock_needed)
11829 {
11830         clock_t                 timer;
11831         mbx_cmd_t               mc;
11832         mbx_cmd_t               *mcp = &mc;
11833         int                     rval = QL_SUCCESS;
11834         ql_adapter_state_t      *ha = vha->pha;

11836         QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

11838         if (CFG_IST(ha, CFG_CTRL_8021)) {
11839                 EL(ha, "8021 not supported\n");
11840                 return (QL_NOT_SUPPORTED);
11841         }

11843         QL_DUMP_LOCK(ha);

11845         if (ha->ql_dump_state & QL_DUMPING ||
11846             (ha->ql_dump_state & QL_DUMP_VALID &&
11847             !(ha->ql_dump_state & QL_DUMP_UPLOADED))) {
11848                 EL(ha, "dump already done, qds=%x\n", ha->ql_dump_state);
11849                 QL_DUMP_UNLOCK(ha);
11850                 return (QL_DATA_EXISTS);
11851         }

11853         ha->ql_dump_state &= ~(QL_DUMP_VALID | QL_DUMP_UPLOADED);
11854         ha->ql_dump_state |= QL_DUMPING;

11856         QL_DUMP_UNLOCK(ha);

11858         if (CFG_IST(ha, CFG_ENABLE_FWEXTTRACE)) {

11860                 /* Insert Time Stamp */
11861                 rval = ql_fw_etrace(ha, &ha->fwexttracebuf,
```

```
11862                       FTO_INSERT_TIME_STAMP);
11863                   if (rval != QL_SUCCESS) {
11864                           EL(ha, "f/w extended trace insert"
11865                               "time stamp failed: %xh\n", rval);
11866                   }
11867           }

11869           if (lock_needed == TRUE) {
11870                   /* Acquire mailbox register lock. */
11871                   MBX_REGISTER_LOCK(ha);
11872                   timer = drv_sectohz(ha->mcp->timeout + 2);
11872                   timer = (ha->mcp->timeout + 2) * drv_usectohz(1000000);

11874                   /* Check for mailbox available, if not wait for signal. */
11875                   while (ha->mailbox_flags & MBX_BUSY_FLG) {
11876                           ha->mailbox_flags = (uint8_t)
11877                               (ha->mailbox_flags | MBX_WANT_FLG);

11879                           /* 30 seconds from now */
11880                           if (cv_reltimedwait(&ha->cv_mbx_wait, &ha->mbx_mutex,
11881                               timer, TR_CLOCK_TICK) == -1) {
11882                                   /*
11883                                    * The timeout time 'timer' was
11884                                    * reached without the condition
11885                                    * being signaled.
11886                                    */

11888                                   /* Release mailbox register lock. */
11889                                   MBX_REGISTER_UNLOCK(ha);

11891                                   EL(ha, "failed, rval = %xh\n",
11892                                       QL_FUNCTION_TIMEOUT);
11893                                   return (QL_FUNCTION_TIMEOUT);
11894                           }
11895                   }

11897                   /* Set busy flag. */
11898                   ha->mailbox_flags = (uint8_t)
11899                       (ha->mailbox_flags | MBX_BUSY_FLG);
11900                   mcp->timeout = 120;
11901                   ha->mcp = mcp;

11903                   /* Release mailbox register lock. */
11904                   MBX_REGISTER_UNLOCK(ha);
11905           }

11907           /* Free previous dump buffer. */
11908           if (ha->ql_dump_ptr != NULL) {
11909                   kmem_free(ha->ql_dump_ptr, ha->ql_dump_size);
11910                   ha->ql_dump_ptr = NULL;
11911           }

11913           if (CFG_IST(ha, CFG_CTRL_2422)) {
11914                   ha->ql_dump_size = (uint32_t)(sizeof (ql_24xx_fw_dump_t) +
11915                       ha->fw_ext_memory_size);
11916           } else if (CFG_IST(ha, CFG_CTRL_25XX)) {
11917                   ha->ql_dump_size = (uint32_t)(sizeof (ql_25xx_fw_dump_t) +
11918                       ha->fw_ext_memory_size);
11919           } else if (CFG_IST(ha, CFG_CTRL_81XX)) {
11920                   ha->ql_dump_size = (uint32_t)(sizeof (ql_81xx_fw_dump_t) +
11921                       ha->fw_ext_memory_size);
11922           } else {
11923                   ha->ql_dump_size = sizeof (ql_fw_dump_t);
11924           }

11926           if ((ha->ql_dump_ptr = kmem_zalloc(ha->ql_dump_size, KM_NOSLEEP)) ==
```



```
11927                   NULL) {
11928                       rval = QL_MEMORY_ALLOC_FAILED;
11929           } else {
11930                   if (CFG_IST(ha, (CFG_CTRL_2300 | CFG_CTRL_6322))) {
11931                           rval = ql_2300_binary_fw_dump(ha, ha->ql_dump_ptr);
11932                   } else if (CFG_IST(ha, CFG_CTRL_81XX)) {
11933                           rval = ql_81xx_binary_fw_dump(ha, ha->ql_dump_ptr);
11934                   } else if (CFG_IST(ha, CFG_CTRL_25XX)) {
11935                           rval = ql_25xx_binary_fw_dump(ha, ha->ql_dump_ptr);
11936                   } else if (CFG_IST(ha, CFG_CTRL_2422)) {
11937                           rval = ql_24xx_binary_fw_dump(ha, ha->ql_dump_ptr);
11938                   } else {
11939                           rval = ql_2200_binary_fw_dump(ha, ha->ql_dump_ptr);
11940                   }
11941           }

11943           /* Reset ISP chip. */
11944           ql_reset_chip(ha);

11946           QL_DUMP_LOCK(ha);

11948           if (rval != QL_SUCCESS) {
11949                   if (ha->ql_dump_ptr != NULL) {
11950                           kmem_free(ha->ql_dump_ptr, ha->ql_dump_size);
11951                           ha->ql_dump_ptr = NULL;
11952                   }
11953                   ha->ql_dump_state &= ~(QL_DUMPING | QL_DUMP_VALID |
11954                       QL_DUMP_UPLOADED);
11955                   EL(ha, "failed, rval = %xh\n", rval);
11956           } else {
11957                   ha->ql_dump_state &= ~(QL_DUMPING | QL_DUMP_UPLOADED);
11958                   ha->ql_dump_state |= QL_DUMP_VALID;
11959                   EL(ha, "done\n");
11960           }

11962           QL_DUMP_UNLOCK(ha);

11964           return (rval);
11965   }
______unchanged_portion_omitted_

15904   static int
15905   ql_suspend_adapter(ql_adapter_state_t *ha)
15906   {
15907           clock_t timer = drv_sectohz(32);
15907           clock_t timer = 32 * drv_usectohz(1000000);

15909           QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

15911           /*
15912            * First we will claim mbox ownership so that no
15913            * thread using mbox hangs when we disable the
15914            * interrupt in the middle of it.
15915            */
15916           MBX_REGISTER_LOCK(ha);

15918           /* Check for mailbox available, if not wait for signal. */
15919           while (ha->mailbox_flags & MBX_BUSY_FLG) {
15920                   ha->mailbox_flags = (uint8_t)
15921                       (ha->mailbox_flags | MBX_WANT_FLG);

15923                   /* 30 seconds from now */
15924                   if (cv_reltimedwait(&ha->cv_mbx_wait, &ha->mbx_mutex,
15925                       timer, TR_CLOCK_TICK) == -1) {

15927                           /* Release mailbox register lock. */
```

```
15928                           MBX_REGISTER_UNLOCK(ha);
15929                           EL(ha, "failed, Suspend mbox");
15930                           return (QL_FUNCTION_TIMEOUT);
15931                   }
15932           }

15934           /* Set busy flag. */
15935           ha->mailbox_flags = (uint8_t)(ha->mailbox_flags | MBX_BUSY_FLG);
15936           MBX_REGISTER_UNLOCK(ha);

15938           (void) ql_wait_outstanding(ha);

15940           /*
15941            * here we are sure that there will not be any mbox interrupt.
15942            * So, let's make sure that we return back all the outstanding
15943            * cmds as well as internally queued commands.
15944            */
15945           ql_halt(ha, PM_LEVEL_D0);

15947           if (ha->power_level != PM_LEVEL_D3) {
15948                   /* Disable ISP interrupts. */
15949                   WRT16_IO_REG(ha, ictrl, 0);
15950           }

15952           ADAPTER_STATE_LOCK(ha);
15953           ha->flags &= ~INTERRUPTS_ENABLED;
15954           ADAPTER_STATE_UNLOCK(ha);

15956           MBX_REGISTER_LOCK(ha);
15957           /* Reset busy status. */
15958           ha->mailbox_flags = (uint8_t)(ha->mailbox_flags & ~MBX_BUSY_FLG);

15960           /* If thread is waiting for mailbox go signal it to start. */
15961           if (ha->mailbox_flags & MBX_WANT_FLG) {
15962                   ha->mailbox_flags = (uint8_t)
15963                       (ha->mailbox_flags & ~MBX_WANT_FLG);
15964                   cv_broadcast(&ha->cv_mbx_wait);
15965           }
15966           /* Release mailbox register lock. */
15967           MBX_REGISTER_UNLOCK(ha);

15969           QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);

15971           return (QL_SUCCESS);
15972 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  112981 Wed Aug 19 07:25:01 2015
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_mbx.c
XXXX introduce drv_sectohz
**********************************************************
```

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

/* Copyright 2010 QLogic Corporation */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#pragma ident   "Copyright 2010 QLogic Corporation; ql_mbx.c"

/*
 * ISP2xxx Solaris Fibre Channel Adapter (FCA) driver source file.
 *
 * ***********************************************************************
 * *                                                                     **
 * *                            NOTICE                                   **
 * *            COPYRIGHT (C) 1996-2010 QLOGIC CORPORATION               **
 * *                    ALL RIGHTS RESERVED                              **
 * *                                                                     **
 * ***********************************************************************
 *
 */

#include <ql_apps.h>
#include <ql_api.h>
#include <ql_debug.h>
#include <ql_iocb.h>
#include <ql_isr.h>
#include <ql_mbx.h>
#include <ql_xioctl.h>

/*
 * Local data
 */

/*
 * Local prototypes
 */
static int ql_mailbox_command(ql_adapter_state_t *, mbx_cmd_t *);
static int ql_task_mgmt_iocb(ql_adapter_state_t *, ql_tgt_t *, uint16_t,
    uint32_t, uint16_t);
static int ql_abort_cmd_iocb(ql_adapter_state_t *, ql_srb_t *);
```



```
static int ql_setup_mbox_dma_transfer(ql_adapter_state_t *, dma_mem_t *,
    caddr_t, uint32_t);
static int ql_setup_mbox_dma_resources(ql_adapter_state_t *, dma_mem_t *,
    uint32_t);
static void ql_setup_mbox_dma_data(dma_mem_t *, caddr_t);
static void ql_get_mbox_dma_data(dma_mem_t *, caddr_t);

/*
 * ql_mailbox_command
 *      Issue mailbox command and waits for completion.
 *
 * Input:
 *      ha = adapter state pointer.
 *      mcp = mailbox command parameter structure pointer.
 *
 * Returns:
 *      ql local function return status code.
 *
 * Context:
 *      Kernel context.
 */
static int
ql_mailbox_command(ql_adapter_state_t *vha, mbx_cmd_t *mcp)
{
        uint16_t                cnt;
        uint32_t                data;
        clock_t                 timer, cv_stat;
        int                     rval;
        uint32_t                set_flags = 0;
        uint32_t                reset_flags = 0;
        ql_adapter_state_t      *ha = vha->pha;
        int                     mbx_cmd = mcp->mb[0];

        QL_PRINT_3(CE_CONT, "(%d): started, cmd=%xh\n", ha->instance, mbx_cmd);

        /* Acquire mailbox register lock. */
        MBX_REGISTER_LOCK(ha);

        /* Check for mailbox available, if not wait for signal. */
        while (ha->mailbox_flags & MBX_BUSY_FLG ||
            (CFG_IST(ha, CFG_CTRL_8021) &&
            RD32_IO_REG(ha, nx_host_int) & NX_MBX_CMD)) {
                ha->mailbox_flags = (uint8_t)
                    (ha->mailbox_flags | MBX_WANT_FLG);

                if (ha->task_daemon_flags & TASK_DAEMON_POWERING_DOWN) {
                        EL(vha, "failed availability cmd=%xh\n", mcp->mb[0]);
                        MBX_REGISTER_UNLOCK(ha);
                        return (QL_LOCK_TIMEOUT);
                }

                /* Set timeout after command that is running. */
                timer = drv_sectohz(mcp->timeout + 20);
                timer = (mcp->timeout + 20) * drv_usectohz(1000000);
                cv_stat = cv_reltimedwait_sig(&ha->cv_mbx_wait,
                    &ha->pha->mbx_mutex, timer, TR_CLOCK_TICK);
                if (cv_stat == -1 || cv_stat == 0) {
                        /*
                         * The timeout time 'timer' was
                         * reached without the condition
                         * being signaled.
                         */
                        ha->mailbox_flags = (uint8_t)(ha->mailbox_flags &
                            ~MBX_WANT_FLG);
                        cv_broadcast(&ha->cv_mbx_wait);
```

```
127                         /* Release mailbox register lock. */
128                         MBX_REGISTER_UNLOCK(ha);
129
130                         if (cv_stat == 0) {
131                                 EL(vha, "waiting for availability aborted, "
132                                     "cmd=%xh\n", mcp->mb[0]);
133                                 return (QL_ABORTED);
134                         }
135                         EL(vha, "failed availability cmd=%xh\n", mcp->mb[0]);
136                         return (QL_LOCK_TIMEOUT);
137                 }
138         }
139
140         ha->mailbox_flags = (uint8_t)(ha->mailbox_flags | MBX_BUSY_FLG);
141
142         /* Structure pointer for return mailbox registers. */
143         ha->mcp = mcp;
144
145         /* Load mailbox registers. */
146         data = mcp->out_mb;
147         for (cnt = 0; cnt < ha->reg_off->mbox_cnt && data; cnt++) {
148                 if (data & MBX_0) {
149                         WRT16_IO_REG(ha, mailbox_in[cnt], mcp->mb[cnt]);
150                 }
151                 data >>= 1;
152         }
153
154         /* Issue set host interrupt command. */
155         ha->mailbox_flags = (uint8_t)(ha->mailbox_flags & ~MBX_INTERRUPT);
156         if (CFG_IST(ha, CFG_CTRL_8021)) {
157                 WRT32_IO_REG(ha, nx_host_int, NX_MBX_CMD);
158         } else if (CFG_IST(ha, CFG_CTRL_242581)) {
159                 WRT32_IO_REG(ha, hccr, HC24_SET_HOST_INT);
160         } else {
161                 WRT16_IO_REG(ha, hccr, HC_SET_HOST_INT);
162         }
163
164         /* Wait for command to complete. */
165         if (ha->flags & INTERRUPTS_ENABLED &&
166             !(ha->task_daemon_flags & (TASK_THREAD_CALLED |
167             TASK_DAEMON_POWERING_DOWN)) &&
168             !ddi_in_panic()) {
169                 timer = drv_sectohz(mcp->timeout);
169                 timer = mcp->timeout * drv_usectohz(1000000);
170                 while (!(ha->mailbox_flags & (MBX_INTERRUPT | MBX_ABORT)) &&
171                     !(ha->task_daemon_flags & ISP_ABORT_NEEDED)) {
172
173                         if (cv_reltimedwait(&ha->cv_mbx_intr,
174                             &ha->pha->mbx_mutex, timer, TR_CLOCK_TICK) == -1) {
175                                 /*
176                                  * The timeout time 'timer' was
177                                  * reached without the condition
178                                  * being signaled.
179                                  */
180                                 MBX_REGISTER_UNLOCK(ha);
181                                 while (INTERRUPT_PENDING(ha)) {
182                                         (void) ql_isr((caddr_t)ha);
183                                         INTR_LOCK(ha);
184                                         ha->intr_claimed = B_TRUE;
185                                         INTR_UNLOCK(ha);
186                                 }
187                                 MBX_REGISTER_LOCK(ha);
188                                 break;
189                         }
190                 }
191         } else {
```



```
192                 /* Release mailbox register lock. */
193                 MBX_REGISTER_UNLOCK(ha);
194
195                 /* Acquire interrupt lock. */
196                 for (timer = mcp->timeout * 100; timer; timer--) {
197                         /* Check for pending interrupts. */
198                         while (INTERRUPT_PENDING(ha)) {
199                                 (void) ql_isr((caddr_t)ha);
200                                 INTR_LOCK(ha);
201                                 ha->intr_claimed = B_TRUE;
202                                 INTR_UNLOCK(ha);
203                                 if (ha->mailbox_flags &
204                                     (MBX_INTERRUPT | MBX_ABORT) ||
205                                     ha->task_daemon_flags & ISP_ABORT_NEEDED) {
206                                         break;
207                                 }
208                         }
209                         if (ha->mailbox_flags & (MBX_INTERRUPT | MBX_ABORT) ||
210                             ha->task_daemon_flags & ISP_ABORT_NEEDED) {
211                                 break;
212                         } else if (!ddi_in_panic() && timer % 101 == 0) {
213                                 delay(drv_usectohz(10000));
214                         } else {
215                                 drv_usecwait(10000);
216                         }
217                 }
218
219                 /* Acquire mailbox register lock. */
220                 MBX_REGISTER_LOCK(ha);
221         }
222
223         /* Mailbox command timeout? */
224         if (ha->task_daemon_flags & ISP_ABORT_NEEDED ||
225             ha->mailbox_flags & MBX_ABORT) {
226                 rval = QL_ABORTED;
227         } else if ((ha->mailbox_flags & MBX_INTERRUPT) == 0) {
228                 if (!CFG_IST(ha, CFG_CTRL_8021)) {
229                         if (CFG_IST(ha, CFG_DUMP_MAILBOX_TIMEOUT)) {
230                                 (void) ql_binary_fw_dump(ha, FALSE);
231                         }
232                         EL(vha, "command timeout, isp_abort_needed\n");
233                         set_flags |= ISP_ABORT_NEEDED;
234                 }
235                 rval = QL_FUNCTION_TIMEOUT;
236         } else {
237                 ha->mailbox_flags = (uint8_t)
238                     (ha->mailbox_flags & ~MBX_INTERRUPT);
239                 /*
240                  * This is the expected completion path so
241                  * return the actual mbx cmd completion status.
242                  */
243                 rval = mcp->mb[0];
244         }
245
246         /*
247          * Clear outbound to risc mailbox registers per spec. The exception
248          * is on 2200 mailbox 4 and 5 affect the req and resp que indexes
249          * so avoid writing them.
250          */
251         if (ha->cfg_flags & CFG_CTRL_2200) {
252                 data = ((mcp->out_mb & ~(MBX_4 | MBX_5)) >> 1);
253         } else {
254                 data = (mcp->out_mb >> 1);
255         }
256         for (cnt = 1; cnt < ha->reg_off->mbox_cnt && data; cnt++) {
257                 if (data & MBX_0) {
```

```
258                         WRT16_IO_REG(ha, mailbox_in[cnt], (uint16_t)0);
259                 }
260                 data >>= 1;
261         }

263         /* Reset busy status. */
264         ha->mailbox_flags = (uint8_t)(ha->mailbox_flags &
265             ~(MBX_BUSY_FLG | MBX_ABORT));
266         ha->mcp = NULL;

268         /* If thread is waiting for mailbox go signal it to start. */
269         if (ha->mailbox_flags & MBX_WANT_FLG) {
270                 ha->mailbox_flags = (uint8_t)(ha->mailbox_flags &
271                     ~MBX_WANT_FLG);
272                 cv_broadcast(&ha->cv_mbx_wait);
273         }

275         /* Release mailbox register lock. */
276         MBX_REGISTER_UNLOCK(ha);

278         if (set_flags != 0 || reset_flags != 0) {
279                 ql_awaken_task_daemon(ha, NULL, set_flags, reset_flags);
280         }

282         if (rval != QL_SUCCESS) {
283                 EL(vha, "%s failed, rval=%xh, mcp->mb[0]=%xh\n",
284                     mbx_cmd_text(mbx_cmd), rval, mcp->mb[0]);
285         } else {
286                 /*EMPTY*/
287                 QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);
288         }

290         return (rval);
291 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50826 Wed Aug 19 07:25:02 2015
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_nx.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1116 static int
1117 ql_8021_wait_flash_done(ql_adapter_state_t *ha)
1118 {
1119         clock_t         timer;
1120         uint32_t        status;

1122         for (timer = drv_sectohz(30); timer; timer--) {
1122         for (timer = 30 * drv_usectohz(1000000); timer; timer--) {
1123                 ql_8021_wr_32(ha, UNM_ROMUSB_ROM_ABYTE_CNT, 0);
1124                 ql_8021_wr_32(ha, UNM_ROMUSB_ROM_INSTR_OPCODE,
1125                     UNM_ROMUSB_ROM_RDSR_INSTR);
1126                 if (ql_8021_wait_rom_done(ha)) {
1127                         EL(ha, "Error waiting for rom done2\n");
1128                         return (-1);
1129                 }

1131                 /* Get status. */
1132                 ql_8021_rd_32(ha, UNM_ROMUSB_ROM_RDATA, &status);
1133                 if (!(status & BIT_0)) {
1134                         return (0);
1135                 }
1136                 delay(1);
1137         }

1139         EL(ha, "timeout status=%x\n", status);
1140         return (-1);
1141 }
______unchanged_portion_omitted_

1604 static int
1605 ql_8021_init_p3p(ql_adapter_state_t *ha)
1606 {
1607         uint32_t        data;

1609         /* ??? */
1610         ql_8021_wr_32(ha, UNM_PORT_MODE_ADDR, UNM_PORT_MODE_AUTO_NEG);
1611         delay(drv_sectohz(1));
1611         delay(drv_usectohz(1000000));

1613         /* CAM RAM Cold Boot Register */
1614         ql_8021_rd_32(ha, UNM_RAM_COLD_BOOT, &data);
1615         if (data == 0x55555555) {
1616                 ql_8021_rd_32(ha, UNM_ROMUSB_GLB_SW_RESET, &data);
1617                 if (data != 0x80000f) {
1618                         EL(ha, "CRB_UNM_GLB_SW_RST=%x exit\n", data);
1619                         return (-1);
1620                 }
1621                 ql_8021_wr_32(ha, UNM_RAM_COLD_BOOT, 0);
1622         }
1623         ql_8021_rd_32(ha, UNM_ROMUSB_GLB_PEGTUNE_DONE, &data);
1624         data |= 1;
1625         ql_8021_wr_32(ha, UNM_ROMUSB_GLB_PEGTUNE_DONE, data);

1627         /*
1628          * ???
1629          * data = ha->pci_bus_addr | BIT_31;
1630          * ql_8021_wr_32(ha, UNM_BUS_DEV_NO, data);
1631          */
```



```
1633         return (0);
1634 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34530 Wed Aug 19 07:25:02 2015
new/usr/src/uts/common/io/fibre-channel/fca/qlge/qlge_mpi.c
XXXX introduce drv_sectohz
**********************************************************
_____unchanged_portion_omitted_
 285 /*
 286  * Send mail box command (upto 16) to MPI Firmware
 287  * and wait for MPI mailbox completion response which
 288  * is saved in interrupt. Thus, this function can only
 289  * be used after interrupt is enabled.
 290  * Must hold MBX mutex before calling this function
 291  */
 292 static int
 293 ql_issue_mailbox_cmd_and_wait_rsp(qlge_t *qlge, mbx_cmd_t *mbx_cmd)
 294 {
 295         int rtn_val = DDI_FAILURE;
 296         clock_t timer;
 297         int i;
 298         int done = 0;

 300         if (mbx_cmd == NULL)
 301                 goto err;

 303         ASSERT(mutex_owned(&qlge->mbx_mutex));

 305         /* if interrupts are not enabled, poll when results are available */
 306         if (!(qlge->flags & INTERRUPTS_ENABLED)) {
 307                 rtn_val = ql_issue_mailbox_cmd_and_poll_rsp(qlge, mbx_cmd,
 308                     &qlge->received_mbx_cmds);
 309                 if (rtn_val == DDI_SUCCESS) {
 310                         for (i = 0; i < NUM_MAILBOX_REGS; i++)
 311                                 mbx_cmd->mb[i] = qlge->received_mbx_cmds.mb[i];
 312                 }
 313         } else {
 314                 rtn_val = ql_issue_mailbox_cmd(qlge, mbx_cmd);
 315                 if (rtn_val != DDI_SUCCESS) {
 316                         cmn_err(CE_WARN, "%s(%d) ql_issue_mailbox_cmd failed",
 317                             __func__, qlge->instance);
 318                         goto err;
 319                 }
 320                 qlge->mbx_wait_completion = 1;
 321                 while (!done && qlge->mbx_wait_completion && !ddi_in_panic()) {
 322                         /* default 5 seconds from now to timeout */
 323                         timer = ddi_get_lbolt();
 324                         if (mbx_cmd->timeout) {
 325                                 timer += drv_sectohz(mbx_cmd->timeout);
 325                                 timer +=
 326                                     mbx_cmd->timeout * drv_usectohz(1000000);
 326                         } else {
 327                                 timer += drv_sectohz(5);
 328                                 timer += 5 * drv_usectohz(1000000);
 328                         }
 329                         if (cv_timedwait(&qlge->cv_mbx_intr, &qlge->mbx_mutex,
 330                             timer) == -1) {
 331                                 /*
 332                                  * The timeout time 'timer' was
 333                                  * reached or expired without the condition
 334                                  * being signaled.
 335                                  */
 336                                 cmn_err(CE_WARN, "%s(%d) Wait for Mailbox cmd "
 337                                     "complete timeout.",
 338                                     __func__, qlge->instance);
 339                                 rtn_val = DDI_FAILURE;
 340                                 done = 1;
 341                         } else {
```



```
 342                                 QL_PRINT(DBG_MBX,
 343                                     ("%s(%d) mailbox completion signal received"
 344                                     " \n", __func__, qlge->instance));
 345                                 for (i = 0; i < NUM_MAILBOX_REGS; i++) {
 346                                         mbx_cmd->mb[i] =
 347                                             qlge->received_mbx_cmds.mb[i];
 348                                 }
 349                                 rtn_val = DDI_SUCCESS;
 350                                 done = 1;
 351                         }
 352                 }
 353         }
 354 err:
 355         return (rtn_val);
 356 }
______unchanged_portion_omitted_
```

```
**********************************************************
  159937 Wed Aug 19 07:25:02 2015
new/usr/src/uts/common/io/fibre-channel/impl/fctl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


/*
 * fc_ulp_add:
 *              Add a ULP module
 *
 * Return Codes:
 *              FC_ULP_SAMEMODULE
 *              FC_SUCCESS
 *              FC_FAILURE
 *
 *   fc_ulp_add  prints  a warning message if there is  already a
 *   similar ULP type  attached and this is unlikely to change as
 *   we trudge along.  Further, this  function  returns a failure
 *   code if the same  module  attempts to add more than once for
 *   the same FC-4 type.
 */
int
fc_ulp_add(fc_ulp_modinfo_t *ulp_info)
{
        fc_ulp_module_t *mod;
        fc_ulp_module_t *prev;
        job_request_t   *job;
        fc_ulp_list_t   *new;
        fc_fca_port_t   *fca_port;
        int             ntry = 0;

        ASSERT(ulp_info != NULL);

        /*
         * Make sure ulp_rev matches fctl version.
         * Whenever non-private data structure or non-static interface changes,
         * we should use an increased FCTL_ULP_MODREV_# number here and in all
         * ulps to prevent version mismatch.
         */
        if (ulp_info->ulp_rev != FCTL_ULP_MODREV_4) {
                cmn_err(CE_WARN, "fctl: ULP %s version mismatch;"
                    " ULP %s would not be loaded", ulp_info->ulp_name,
                    ulp_info->ulp_name);
                return (FC_BADULP);
        }

        new = kmem_zalloc(sizeof (*new), KM_SLEEP);
        ASSERT(new != NULL);

        mutex_enter(&fctl_ulp_list_mutex);
        new->ulp_info = ulp_info;
        if (fctl_ulp_list != NULL) {
                new->ulp_next = fctl_ulp_list;
        }
        fctl_ulp_list = new;
        mutex_exit(&fctl_ulp_list_mutex);

        while (rw_tryenter(&fctl_ulp_lock, RW_WRITER) == 0) {
                delay(drv_sectohz(1));
                delay(drv_usectohz(1000000));
                if (ntry++ > FC_ULP_ADD_RETRY_COUNT) {
                        fc_ulp_list_t   *list;
                        fc_ulp_list_t   *last;
                        mutex_enter(&fctl_ulp_list_mutex);
```



```
                        for (last = NULL, list = fctl_ulp_list; list != NULL;
                            list = list->ulp_next) {
                                if (list->ulp_info == ulp_info) {
                                        break;
                                }
                                last = list;
                        }

                        if (list) {
                                if (last) {
                                        last->ulp_next = list->ulp_next;
                                } else {
                                        fctl_ulp_list = list->ulp_next;
                                }
                                kmem_free(list, sizeof (*list));
                        }
                        mutex_exit(&fctl_ulp_list_mutex);
                        cmn_err(CE_WARN, "fctl: ULP %s unable to load",
                            ulp_info->ulp_name);
                        return (FC_FAILURE);
                }
        }

        for (mod = fctl_ulp_modules, prev = NULL; mod; mod = mod->mod_next) {
                ASSERT(mod->mod_info != NULL);

                if (ulp_info == mod->mod_info &&
                    ulp_info->ulp_type == mod->mod_info->ulp_type) {
                        rw_exit(&fctl_ulp_lock);
                        return (FC_ULP_SAMEMODULE);
                }

                if (ulp_info->ulp_type == mod->mod_info->ulp_type) {
                        cmn_err(CE_NOTE, fctl_greeting, ulp_info->ulp_name,
                            ulp_info->ulp_type);
                }
                prev = mod;
        }

        mod = kmem_zalloc(sizeof (*mod), KM_SLEEP);
        mod->mod_info = ulp_info;
        mod->mod_next = NULL;

        if (prev) {
                prev->mod_next = mod;
        } else {
                fctl_ulp_modules = mod;
        }

        /*
         * Schedule a job to each port's job_handler
         * thread to attach their ports with this ULP.
         */
        mutex_enter(&fctl_port_lock);
        for (fca_port = fctl_fca_portlist; fca_port != NULL;
            fca_port = fca_port->port_next) {
                job = fctl_alloc_job(JOB_ATTACH_ULP, JOB_TYPE_FCTL_ASYNC,
                    NULL, NULL, KM_SLEEP);

                fctl_enque_job(fca_port->port_handle, job);
        }
        mutex_exit(&fctl_port_lock);

        rw_exit(&fctl_ulp_lock);

        return (FC_SUCCESS);
```

```
 728 }
______unchanged_portion_omitted_
```

```
**********************************************************
  387988 Wed Aug 19 07:25:02 2015
new/usr/src/uts/common/io/fibre-channel/impl/fp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


/*
 * fp_detach:
 *
 * If a ULP fails to handle cmd request converse of
 * cmd is invoked for ULPs that previously succeeded
 * cmd request.
 */
static int
fp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        int                     rval = DDI_FAILURE;
        fc_local_port_t         *port;
        fc_attach_cmd_t         converse;
        uint8_t                 cnt;

        if ((port = ddi_get_soft_state(fp_driver_softstate,
            ddi_get_instance(dip))) == NULL) {
                return (DDI_FAILURE);
        }

        mutex_enter(&port->fp_mutex);

        if (port->fp_ulp_attach) {
                mutex_exit(&port->fp_mutex);
                return (DDI_FAILURE);
        }

        switch (cmd) {
        case DDI_DETACH:
                if (port->fp_task != FP_TASK_IDLE) {
                        mutex_exit(&port->fp_mutex);
                        return (DDI_FAILURE);
                }

                /* Let's attempt to quit the job handler gracefully */
                port->fp_soft_state |= FP_DETACH_INPROGRESS;

                mutex_exit(&port->fp_mutex);
                converse = FC_CMD_ATTACH;
                if (fctl_detach_ulps(port, FC_CMD_DETACH,
                    &modlinkage) != FC_SUCCESS) {
                        mutex_enter(&port->fp_mutex);
                        port->fp_soft_state &= ~FP_DETACH_INPROGRESS;
                        mutex_exit(&port->fp_mutex);
                        rval = DDI_FAILURE;
                        break;
                }

                mutex_enter(&port->fp_mutex);
                for (cnt = 0; (port->fp_job_head) && (cnt < fp_cmd_wait_cnt);
                    cnt++) {
                        mutex_exit(&port->fp_mutex);
                        delay(drv_sectohz(1));
                        delay(drv_usectohz(1000000));
                        mutex_enter(&port->fp_mutex);
                }

                if (port->fp_job_head) {
```



```
                        mutex_exit(&port->fp_mutex);
                        rval = DDI_FAILURE;
                        break;
                }
                mutex_exit(&port->fp_mutex);

                rval = fp_detach_handler(port);
                break;

        case DDI_SUSPEND:
                mutex_exit(&port->fp_mutex);
                converse = FC_CMD_RESUME;
                if (fctl_detach_ulps(port, FC_CMD_SUSPEND,
                    &modlinkage) != FC_SUCCESS) {
                        rval = DDI_FAILURE;
                        break;
                }
                if ((rval = fp_suspend_handler(port)) != DDI_SUCCESS) {
                        (void) callb_generic_cpr(&port->fp_cpr_info,
                            CB_CODE_CPR_RESUME);
                }
                break;

        default:
                mutex_exit(&port->fp_mutex);
                break;
        }

        /*
         * Use softint to perform reattach.  Mark fp_ulp_attach so we
         * don't attempt to do this repeatedly on behalf of some persistent
         * caller.
         */
        if (rval != DDI_SUCCESS) {
                mutex_enter(&port->fp_mutex);
                port->fp_ulp_attach = 1;

                /*
                 * If the port is in the low power mode then there is
                 * possibility that fca too could be in low power mode.
                 * Try to raise the power before calling attach ulps.
                 */

                if ((port->fp_soft_state & FP_SOFT_POWER_DOWN) &&
                    (!(port->fp_soft_state & FP_SOFT_NO_PMCOMP))) {
                        mutex_exit(&port->fp_mutex);
                        (void) pm_raise_power(port->fp_port_dip,
                            FP_PM_COMPONENT, FP_PM_PORT_UP);
                } else {
                        mutex_exit(&port->fp_mutex);
                }


                fp_attach_ulps(port, converse);

                mutex_enter(&port->fp_mutex);
                while (port->fp_ulp_attach) {
                        cv_wait(&port->fp_attach_cv, &port->fp_mutex);
                }

                port->fp_soft_state &= ~FP_DETACH_INPROGRESS;

                /*
                 * Mark state as detach failed so asynchronous ULP attach
                 * events (downstream, not the ones we're initiating with
                 * the call to fp_attach_ulps) are not honored.  We're
```

```
 512                 * really still in pending detach.
 513                 */
 514                port->fp_soft_state |= FP_DETACH_FAILED;

 516                mutex_exit(&port->fp_mutex);
 517        }

 519        return (rval);
 520 }
```

*unchanged_portion_omitted_*

```
1453 /*
1454  * At this time, there shouldn't be any I/O requests on this port.
1455  * But the unsolicited callbacks from the underlying FCA port need
1456  * to be handled very carefully. The steps followed to handle the
1457  * DDI_DETACH are:
1458  *      +       Grab the port driver mutex, check if the unsolicited
1459  *              callback is currently under processing. If true, fail
1460  *              the DDI_DETACH request by printing a message; If false
1461  *              mark the DDI_DETACH as under progress, so that any
1462  *              further unsolicited callbacks get bounced.
1463  *      +       Perform PRLO/LOGO if necessary, cleanup all the data
1464  *              structures.
1465  *      +       Get the job_handler thread to gracefully exit.
1466  *      +       Unregister callbacks with the FCA port.
1467  *      +       Now that some peace is found, notify all the ULPs of
1468  *              DDI_DETACH request (using ulp_port_detach entry point)
1469  *      +       Free all mutexes, semaphores, conditional variables.
1470  *      +       Free the soft state, return success.
1471  *
1472  * Important considerations:
1473  *              Port driver de-registers state change and unsolicited
1474  *              callbacks before taking up the task of notifying ULPs
1475  *              and performing PRLO and LOGOs.
1476  *
1477  *              A port may go offline at the time PRLO/LOGO is being
1478  *              requested. It is expected of all FCA drivers to fail
1479  *              such requests either immediately with a FC_OFFLINE
1480  *              return code to fc_fca_transport() or return the packet
1481  *              asynchronously with pkt state set to FC_PKT_PORT_OFFLINE
1482  */
1483 static int
1484 fp_detach_handler(fc_local_port_t *port)
1485 {
1486         job_request_t   *job;
1487         uint32_t        delay_count;
1488         fc_orphan_t     *orp, *tmporp;

1490         /*
1491          * In a Fabric topology with many host ports connected to
1492          * a switch, another detaching instance of fp might have
1493          * triggered a LOGO (which is an unsolicited request to
1494          * this instance). So in order to be able to successfully
1495          * detach by taking care of such cases a delay of about
1496          * 30 seconds is introduced.
1497          */
1498         delay_count = 0;
1499         mutex_enter(&port->fp_mutex);
1500         if (port->fp_out_fpcmds != 0) {
1501                 /*
1502                  * At this time we can only check fp internal commands, because
1503                  * sd/ssd/scsi_vhci should have finsihed all their commands,
1504                  * fcp/fcip/fcsm should have finished all their commands.
1505                  *
1506                  * It seems that all fp internal commands are asynchronous now.
```



```
1507                  */
1508                 port->fp_soft_state &= ~FP_DETACH_INPROGRESS;
1509                 mutex_exit(&port->fp_mutex);

1511                 cmn_err(CE_WARN, "fp(%d): %d fp_cmd(s) is/are in progress"
1512                     " Failing detach", port->fp_instance, port->fp_out_fpcmds);
1513                 return (DDI_FAILURE);
1514         }

1516         while ((port->fp_soft_state &
1517             (FP_SOFT_IN_STATEC_CB | FP_SOFT_IN_UNSOL_CB)) &&
1518             (delay_count < 30)) {
1519                 mutex_exit(&port->fp_mutex);
1520                 delay_count++;
1521                 delay(drv_sectohz(1));
1521                 delay(drv_usectohz(1000000));
1522                 mutex_enter(&port->fp_mutex);
1523         }

1525         if (port->fp_soft_state &
1526             (FP_SOFT_IN_STATEC_CB | FP_SOFT_IN_UNSOL_CB)) {
1527                 port->fp_soft_state &= ~FP_DETACH_INPROGRESS;
1528                 mutex_exit(&port->fp_mutex);

1530                 cmn_err(CE_WARN, "fp(%d): FCA callback in progress: "
1531                     " Failing detach", port->fp_instance);
1532                 return (DDI_FAILURE);
1533         }

1535         port->fp_soft_state |= FP_SOFT_IN_DETACH;
1536         port->fp_soft_state &= ~FP_DETACH_INPROGRESS;
1537         mutex_exit(&port->fp_mutex);

1539         /*
1540          * If we're powered down, we need to raise power prior to submitting
1541          * the JOB_PORT_SHUTDOWN job.  Otherwise, the job handler will never
1542          * process the shutdown job.
1543          */
1544         if (fctl_busy_port(port) != 0) {
1545                 cmn_err(CE_WARN, "fp(%d): fctl_busy_port failed",
1546                     port->fp_instance);
1547                 mutex_enter(&port->fp_mutex);
1548                 port->fp_soft_state &= ~FP_SOFT_IN_DETACH;
1549                 mutex_exit(&port->fp_mutex);
1550                 return (DDI_FAILURE);
1551         }

1553         /*
1554          * This will deallocate data structs and cause the "job" thread
1555          * to exit, in preparation for DDI_DETACH on the instance.
1556          * This can sleep for an arbitrary duration, since it waits for
1557          * commands over the wire, timeout(9F) callbacks, etc.
1558          *
1559          * CAUTION: There is still a race here, where the "job" thread
1560          * can still be executing code even tho the fctl_jobwait() call
1561          * below has returned to us.  In theory the fp driver could even be
1562          * modunloaded even tho the job thread isn't done executing.
1563          * without creating the race condition.
1564          */
1565         job = fctl_alloc_job(JOB_PORT_SHUTDOWN, 0, NULL,
1566             (opaque_t)port, KM_SLEEP);
1567         fctl_enque_job(port, job);
1568         fctl_jobwait(job);
1569         fctl_dealloc_job(job);
```

```
1572         (void) pm_lower_power(port->fp_port_dip, FP_PM_COMPONENT,
1573             FP_PM_PORT_DOWN);

1575         if (port->fp_taskq) {
1576                 taskq_destroy(port->fp_taskq);
1577         }

1579         ddi_prop_remove_all(port->fp_port_dip);

1581         ddi_remove_minor_node(port->fp_port_dip, NULL);

1583         fctl_remove_port(port);

1585         fp_free_pkt(port->fp_els_resp_pkt);

1587         if (port->fp_ub_tokens) {
1588                 if (fc_ulp_ubfree(port, port->fp_ub_count,
1589                     port->fp_ub_tokens) != FC_SUCCESS) {
1590                         cmn_err(CE_WARN, "fp(%d): couldn't free "
1591                             " unsolicited buffers", port->fp_instance);
1592                 }
1593                 kmem_free(port->fp_ub_tokens,
1594                     sizeof (*port->fp_ub_tokens) * port->fp_ub_count);
1595                 port->fp_ub_tokens = NULL;
1596         }

1598         if (port->fp_pkt_cache != NULL) {
1599                 kmem_cache_destroy(port->fp_pkt_cache);
1600         }

1602         port->fp_fca_tran->fca_unbind_port(port->fp_fca_handle);

1604         mutex_enter(&port->fp_mutex);
1605         if (port->fp_did_table) {
1606                 kmem_free(port->fp_did_table, did_table_size *
1607                     sizeof (struct d_id_hash));
1608         }

1610         if (port->fp_pwwn_table) {
1611                 kmem_free(port->fp_pwwn_table, pwwn_table_size *
1612                     sizeof (struct pwwn_hash));
1613         }
1614         orp = port->fp_orphan_list;
1615         while (orp) {
1616                 tmporp = orp;
1617                 orp = orp->orp_next;
1618                 kmem_free(tmporp, sizeof (*orp));
1619         }

1621         mutex_exit(&port->fp_mutex);

1623         fp_log_port_event(port, ESC_SUNFC_PORT_DETACH);

1625         mutex_destroy(&port->fp_mutex);
1626         cv_destroy(&port->fp_attach_cv);
1627         cv_destroy(&port->fp_cv);
1628         ddi_soft_state_free(fp_driver_softstate, port->fp_instance);

1630         return (DDI_SUCCESS);
1631 }


1634 /*
1635  * Steps to perform DDI_SUSPEND operation on a FC port
1636  *
1637  *      - If already suspended return DDI_FAILURE
```



```
1638  *      - If already power-suspended return DDI_SUCCESS
1639  *      - If an unsolicited callback or state change handling is in
1640  *          in progress, throw a warning message, return DDI_FAILURE
1641  *      - Cancel timeouts
1642  *      - SUSPEND the job_handler thread (means do nothing as it is
1643  *          taken care of by the CPR frame work)
1644  */
1645 static int
1646 fp_suspend_handler(fc_local_port_t *port)
1647 {
1648         uint32_t        delay_count;

1650         mutex_enter(&port->fp_mutex);

1652         /*
1653          * The following should never happen, but
1654          * let the driver be more defensive here
1655          */
1656         if (port->fp_soft_state & FP_SOFT_SUSPEND) {
1657                 mutex_exit(&port->fp_mutex);
1658                 return (DDI_FAILURE);
1659         }

1661         /*
1662          * If the port is already power suspended, there
1663          * is nothing else to do, So return DDI_SUCCESS,
1664          * but mark the SUSPEND bit in the soft state
1665          * before leaving.
1666          */
1667         if (port->fp_soft_state & FP_SOFT_POWER_DOWN) {
1668                 port->fp_soft_state |= FP_SOFT_SUSPEND;
1669                 mutex_exit(&port->fp_mutex);
1670                 return (DDI_SUCCESS);
1671         }

1673         /*
1674          * Check if an unsolicited callback or state change handling is
1675          * in progress. If true, fail the suspend operation; also throw
1676          * a warning message notifying the failure. Note that Sun PCI
1677          * hotplug spec recommends messages in cases of failure (but
1678          * not flooding the console)
1679          *
1680          * Busy waiting for a short interval (500 millisecond ?) to see
1681          * if the callback processing completes may be another idea. Since
1682          * most of the callback processing involves a lot of work, it
1683          * is safe to just fail the SUSPEND operation. It is definitely
1684          * not bad to fail the SUSPEND operation if the driver is busy.
1685          */
1686         delay_count = 0;
1687         while ((port->fp_soft_state & (FP_SOFT_IN_STATEC_CB |
1688             FP_SOFT_IN_UNSOL_CB)) && (delay_count < 30)) {
1689                 mutex_exit(&port->fp_mutex);
1690                 delay_count++;
1691                 delay(drv_sectohz(1));
1691                 delay(drv_usectohz(1000000));
1692                 mutex_enter(&port->fp_mutex);
1693         }

1695         if (port->fp_soft_state & (FP_SOFT_IN_STATEC_CB |
1696             FP_SOFT_IN_UNSOL_CB)) {
1697                 mutex_exit(&port->fp_mutex);
1698                 cmn_err(CE_WARN, "fp(%d): FCA callback in progress: "
1699                     " Failing suspend", port->fp_instance);
1700                 return (DDI_FAILURE);
1701         }
```

```
1703         /*
1704          * Check of FC port thread is busy
1705          */
1706         if (port->fp_job_head) {
1707                 mutex_exit(&port->fp_mutex);
1708                 FP_TRACE(FP_NHEAD2(9, 0),
1709                     "FC port thread is busy: Failing suspend");
1710                 return (DDI_FAILURE);
1711         }
1712         port->fp_soft_state |= FP_SOFT_SUSPEND;

1714         fp_suspend_all(port);
1715         mutex_exit(&port->fp_mutex);

1717         return (DDI_SUCCESS);
1718 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  207076 Wed Aug 19 07:25:03 2015
new/usr/src/uts/common/io/fibre-channel/ulp/fcip.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1054 /*
1055  * fcip_port_attach_handler : Completes the port attach operation after
1056  * the ulp_port_attach routine has completed its ground work. The job
1057  * of this function among other things is to obtain and handle topology
1058  * specifics, initialize a port, setup broadcast address entries in
1059  * the fcip tables etc. This routine cleans up behind itself on failures.
1060  * Returns FC_SUCCESS or FC_FAILURE.
1061  */
1062 static int
1063 fcip_port_attach_handler(struct fcip *fptr)
1064 {
1065         fcip_port_info_t                *fport = fptr->fcip_port_info;
1066         int                             rval = FC_FAILURE;

1068         ASSERT(fport != NULL);

1070         mutex_enter(&fcip_global_mutex);

1072         FCIP_DEBUG(FCIP_DEBUG_ATTACH, (CE_NOTE,
1073             "fcip module dip: %p instance: %d",
1074             (void *)fcip_module_dip, ddi_get_instance(fptr->fcip_dip)));

1076         if (fcip_module_dip == NULL) {
1077                 clock_t         fcip_lbolt;

1079                 fcip_lbolt = ddi_get_lbolt();
1080                 /*
1081                  * we need to use the fcip devinfo for creating
1082                  * the clone device node, but the fcip attach
1083                  * (from its conf file entry claiming to be a
1084                  * child of pseudo) may not have happened yet.
1085                  * wait here for 10 seconds and fail port attach
1086                  * if the fcip devinfo is not attached yet
1087                  */
1088                 fcip_lbolt += drv_usectohz(FCIP_INIT_DELAY);

1090                 FCIP_DEBUG(FCIP_DEBUG_ATTACH,
1091                     (CE_WARN, "cv_timedwait lbolt %lx", fcip_lbolt));

1093                 (void) cv_timedwait(&fcip_global_cv, &fcip_global_mutex,
1094                     fcip_lbolt);

1096                 if (fcip_module_dip == NULL) {
1097                         mutex_exit(&fcip_global_mutex);

1099                         FCIP_DEBUG(FCIP_DEBUG_ATTACH, (CE_WARN,
1100                                 "fcip attach did not happen"));
1101                         goto port_attach_cleanup;
1102                 }
1103         }

1105         if ((!fcip_minor_node_created) &&
1106             fcip_is_supported_fc_topology(fport->fcipp_topology)) {
1107                 /*
1108                  * Checking for same topologies which are considered valid
1109                  * by fcip_handle_topology(). Dont create a minor node if
1110                  * nothing is hanging off the FC port.
1111                  */
1112                 if (ddi_create_minor_node(fcip_module_dip, "fcip", S_IFCHR,
```



```
1113                     ddi_get_instance(fptr->fcip_dip), DDI_PSEUDO,
1114                     CLONE_DEV) == DDI_FAILURE) {
1115                         mutex_exit(&fcip_global_mutex);
1116                         FCIP_DEBUG(FCIP_DEBUG_ATTACH, (CE_WARN,
1117                             "failed to create minor node for fcip(%d)",
1118                             ddi_get_instance(fptr->fcip_dip)));
1119                         goto port_attach_cleanup;
1120                 }
1121                 fcip_minor_node_created++;
1122         }
1123         mutex_exit(&fcip_global_mutex);

1125         /*
1126          * initialize port for traffic
1127          */
1128         if (fcip_init_port(fptr) != FC_SUCCESS) {
1129                 /* fcip_init_port has already cleaned up its stuff */

1131                 mutex_enter(&fcip_global_mutex);

1133                 if ((fcip_num_instances == 1) &&
1134                     (fcip_minor_node_created == 1)) {
1135                         /* Remove minor node iff this is the last instance */
1136                         ddi_remove_minor_node(fcip_module_dip, NULL);
1137                 }

1139                 mutex_exit(&fcip_global_mutex);

1141                 goto port_attach_cleanup;
1142         }

1144         mutex_enter(&fptr->fcip_mutex);
1145         fptr->fcip_flags &= ~FCIP_ATTACHING;
1146         fptr->fcip_flags |= FCIP_INITED;
1147         fptr->fcip_timeout_ticks = 0;

1149         /*
1150          * start the timeout threads
1151          */
1152         fptr->fcip_timeout_id = timeout(fcip_timeout, fptr,
1153             drv_sectohz(1));
1153             drv_usectohz(1000000));

1155         mutex_exit(&fptr->fcip_mutex);
1156         mutex_enter(&fcip_global_mutex);
1157         fcip_num_attaching--;
1158         ASSERT(fcip_num_attaching >= 0);
1159         mutex_exit(&fcip_global_mutex);
1160         rval = FC_SUCCESS;
1161         return (rval);

1163 port_attach_cleanup:
1164         mutex_enter(&fcip_global_mutex);
1165         (void) fcip_softstate_free(fport);
1166         fcip_num_attaching--;
1167         ASSERT(fcip_num_attaching >= 0);
1168         mutex_exit(&fcip_global_mutex);
1169         rval = FC_FAILURE;
1170         return (rval);
1171 }


1174 /*
1175  * Handler for DDI_RESUME operations. Port must be ready to restart IP
1176  * traffic on resume
1177  */
```

```
1178 static int
1179 fcip_handle_resume(fcip_port_info_t *fport, fc_ulp_port_info_t *port_info,
1180     fc_attach_cmd_t cmd)
1181 {
1182         int             rval = FC_SUCCESS;
1183         struct fcip     *fptr = fport->fcipp_fcip;
1184         struct fcipstr  *tslp;
1185         int             index;


1188         ASSERT(fptr != NULL);

1190         mutex_enter(&fptr->fcip_mutex);

1192         if (cmd == FC_CMD_POWER_UP) {
1193                 fptr->fcip_flags &= ~(FCIP_POWER_DOWN);
1194                 if (fptr->fcip_flags & FCIP_SUSPENDED) {
1195                         mutex_exit(&fptr->fcip_mutex);
1196                         return (FC_SUCCESS);
1197                 }
1198         } else if (cmd == FC_CMD_RESUME) {
1199                 fptr->fcip_flags &= ~(FCIP_SUSPENDED);
1200         } else {
1201                 mutex_exit(&fptr->fcip_mutex);
1202                 return (FC_FAILURE);
1203         }

1205         /*
1206          * set the current port state and topology
1207          */
1208         fport->fcipp_topology = port_info->port_flags;
1209         fport->fcipp_pstate = port_info->port_state;

1211         rw_enter(&fcipstruplock, RW_READER);
1212         for (tslp = fcipstrup; tslp; tslp = tslp->sl_nextp) {
1213                 if (tslp->sl_fcip == fptr) {
1214                         break;
1215                 }
1216         }
1217         rw_exit(&fcipstruplock);

1219         /*
1220          * No active streams on this port
1221          */
1222         if (tslp == NULL) {
1223                 rval = FC_SUCCESS;
1224                 goto done;
1225         }

1227         mutex_enter(&fptr->fcip_rt_mutex);
1228         for (index = 0; index < FCIP_RT_HASH_ELEMS; index++) {
1229                 struct fcip_routing_table       *frp;

1231                 frp = fptr->fcip_rtable[index];
1232                 while (frp) {
1233                         uint32_t                did;
1234                         /*
1235                          * Mark the broadcast RTE available again. It
1236                          * was marked SUSPENDED during SUSPEND.
1237                          */
1238                         did = fcip_get_broadcast_did(fptr);
1239                         if (frp->fcipr_d_id.port_id == did) {
1240                                 frp->fcipr_state = 0;
1241                                 index = FCIP_RT_HASH_ELEMS;
1242                                 break;
1243                         }
```



```
1244                         frp = frp->fcipr_next;
1245                 }
1246         }
1247         mutex_exit(&fptr->fcip_rt_mutex);

1249         /*
1250          * fcip_handle_topology will update the port entries in the
1251          * routing table.
1252          * fcip_handle_topology also takes care of resetting the
1253          * fcipr_state field in the routing table structure. The entries
1254          * were set to RT_INVALID during suspend.
1255          */
1256         fcip_handle_topology(fptr);

1258 done:
1259         /*
1260          * Restart the timeout thread
1261          */
1262         fptr->fcip_timeout_id = timeout(fcip_timeout, fptr,
1263             drv_sectohz(1));
1263             drv_usectohz(1000000));
1264         mutex_exit(&fptr->fcip_mutex);
1265         return (rval);
1266 }
______unchanged_portion_omitted_


7182 /*
7183  * fcip timeout performs 2 operations:
7184  * 1. timeout any packets sent to the FCA for which a callback hasn't
7185  *    happened. If you are wondering why we need a callback since all
7186  *    traffic in FCIP is unidirectional, hence all exchanges are unidirectional
7187  *    but wait, we can only free up the resources after we know the FCA has
7188  *    DMA'ed out the data. pretty obvious eh :)
7189  *
7190  * 2. Retire and routing table entries we marked up for retiring. This is
7191  *    to give the link a chance to recover instead of marking a port down
7192  *    when we have lost all communication with it after a link transition
7193  */
7194 static void
7195 fcip_timeout(void *arg)
7196 {
7197         struct fcip                     *fptr = (struct fcip *)arg;
7198         int                             i;
7199         fcip_pkt_t                      *fcip_pkt;
7200         struct fcip_dest                *fdestp;
7201         int                             index;
7202         struct fcip_routing_table       *frtp;
7203         int                             dispatch_rte_removal = 0;

7205         mutex_enter(&fptr->fcip_mutex);

7207         fptr->fcip_flags |= FCIP_IN_TIMEOUT;
7208         fptr->fcip_timeout_ticks += fcip_tick_incr;

7210         if (fptr->fcip_flags & (FCIP_DETACHED | FCIP_DETACHING | \
7211             FCIP_SUSPENDED | FCIP_POWER_DOWN)) {
7212                 fptr->fcip_flags &= ~(FCIP_IN_TIMEOUT);
7213                 mutex_exit(&fptr->fcip_mutex);
7214                 return;
7215         }

7217         if (fptr->fcip_port_state == FCIP_PORT_OFFLINE) {
7218                 if (fptr->fcip_timeout_ticks > fptr->fcip_mark_offline) {
7219                         fptr->fcip_flags |= FCIP_LINK_DOWN;
7220                 }
```

```
7221        }
7222        if (!fptr->fcip_flags & FCIP_RTE_REMOVING) {
7223                dispatch_rte_removal = 1;
7224        }
7225        mutex_exit(&fptr->fcip_mutex);

7227        /*
7228         * Check if we have any Invalid routing table entries in our
7229         * hashtable we have marked off for deferred removal. If any,
7230         * we can spawn a taskq thread to do the cleanup for us. We
7231         * need to avoid cleanup in the timeout thread since we may
7232         * have to wait for outstanding commands to complete before
7233         * we retire a routing table entry. Also dispatch the taskq
7234         * thread only if we are already do not have a taskq thread
7235         * dispatched.
7236         */
7237        if (dispatch_rte_removal) {
7238                mutex_enter(&fptr->fcip_rt_mutex);
7239                for (index = 0; index < FCIP_RT_HASH_ELEMS; index++) {
7240                        frtp = fptr->fcip_rtable[index];
7241                        while (frtp) {
7242                                if ((frtp->fcipr_state == FCIP_RT_INVALID) &&
7243                                    (fptr->fcip_timeout_ticks >
7244                                    frtp->fcipr_invalid_timeout)) {
7245                                        /*
7246                                         * If we cannot schedule a task thread
7247                                         * let us attempt again on the next
7248                                         * tick rather than call
7249                                         * fcip_rte_remove_deferred() from here
7250                                         * directly since the routine can sleep.
7251                                         */
7252                                        frtp->fcipr_state = FCIP_RT_RETIRED;

7254                                        mutex_enter(&fptr->fcip_mutex);
7255                                        fptr->fcip_flags |= FCIP_RTE_REMOVING;
7256                                        mutex_exit(&fptr->fcip_mutex);

7258                                        if (taskq_dispatch(fptr->fcip_tq,
7259                                            fcip_rte_remove_deferred, fptr,
7260                                            KM_NOSLEEP) == 0) {
7261                                                /*
7262                                                 * failed - so mark the entry
7263                                                 * as invalid again.
7264                                                 */
7265                                                frtp->fcipr_state =
7266                                                    FCIP_RT_INVALID;

7268                                                mutex_enter(&fptr->fcip_mutex);
7269                                                fptr->fcip_flags &=
7270                                                    ~FCIP_RTE_REMOVING;
7271                                                mutex_exit(&fptr->fcip_mutex);
7272                                        }
7273                                }
7274                                frtp = frtp->fcipr_next;
7275                        }
7276                }
7277                mutex_exit(&fptr->fcip_rt_mutex);
7278        }

7280        mutex_enter(&fptr->fcip_dest_mutex);

7282        /*
7283         * Now timeout any packets stuck with the transport/FCA for too long
7284         */
7285        for (i = 0; i < FCIP_DEST_HASH_ELEMS; i++) {
7286                fdestp = fptr->fcip_dest[i];
```



```
7287                while (fdestp != NULL) {
7288                        mutex_enter(&fdestp->fcipd_mutex);
7289                        for (fcip_pkt = fdestp->fcipd_head; fcip_pkt != NULL;
7290                            fcip_pkt = fcip_pkt->fcip_pkt_next) {
7291                                if (fcip_pkt->fcip_pkt_flags &
7292                                    (FCIP_PKT_RETURNED | FCIP_PKT_IN_TIMEOUT |
7293                                    FCIP_PKT_IN_ABORT)) {
7294                                        continue;
7295                                }
7296                                if (fptr->fcip_timeout_ticks >
7297                                    fcip_pkt->fcip_pkt_ttl) {
7298                                        fcip_pkt->fcip_pkt_flags |=
7299                                            FCIP_PKT_IN_TIMEOUT;

7301                                        mutex_exit(&fdestp->fcipd_mutex);
7302                                        if (taskq_dispatch(fptr->fcip_tq,
7303                                            fcip_pkt_timeout, fcip_pkt,
7304                                            KM_NOSLEEP) == 0) {
7305                                                /*
7306                                                 * timeout immediately
7307                                                 */
7308                                                fcip_pkt_timeout(fcip_pkt);
7309                                        }
7310                                        mutex_enter(&fdestp->fcipd_mutex);
7311                                        /*
7312                                         * The linked list is altered because
7313                                         * of one of the following reasons:
7314                                         *      a. Timeout code dequeued a pkt
7315                                         *      b. Pkt completion happened
7316                                         *
7317                                         * So restart the spin starting at
7318                                         * the head again; This is a bit
7319                                         * excessive, but okay since
7320                                         * fcip_timeout_ticks isn't incremented
7321                                         * for this spin, we will skip the
7322                                         * not-to-be-timedout packets quickly
7323                                         */
7324                                        fcip_pkt = fdestp->fcipd_head;
7325                                        if (fcip_pkt == NULL) {
7326                                                break;
7327                                        }
7328                                }
7329                        }
7330                        mutex_exit(&fdestp->fcipd_mutex);
7331                        fdestp = fdestp->fcipd_next;
7332                }
7333        }
7334        mutex_exit(&fptr->fcip_dest_mutex);

7336        /*
7337         * reschedule the timeout thread
7338         */
7339        mutex_enter(&fptr->fcip_mutex);

7341        fptr->fcip_timeout_id = timeout(fcip_timeout, fptr,
7342            drv_sectohz(1));
7342            drv_usectohz(1000000));
7343        fptr->fcip_flags &= ~(FCIP_IN_TIMEOUT);
7344        mutex_exit(&fptr->fcip_mutex);
7345 }
______unchanged_portion_omitted_
```

```
**********************************************************
  435590 Wed Aug 19 07:25:03 2015
new/usr/src/uts/common/io/fibre-channel/ulp/fcp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


3698 /*
3699  * called for ioctls on the transport's devctl interface, and the transport
3700  * has passed it to us
3701  *
3702  * this will only be called for device control ioctls (i.e. hotplugging stuff)
3703  *
3704  * return FC_SUCCESS if we decide to claim the ioctl,
3705  * else return FC_UNCLAIMED
3706  *
3707  * *rval is set iff we decide to claim the ioctl
3708  */
3709 /*ARGSUSED*/
3710 static int
3711 fcp_port_ioctl(opaque_t ulph, opaque_t port_handle, dev_t dev, int cmd,
3712     intptr_t data, int mode, cred_t *credp, int *rval, uint32_t claimed)
3713 {
3714         int                     retval = FC_UNCLAIMED;  /* return value */
3715         struct fcp_port         *pptr = NULL;           /* our soft state */
3716         struct devctl_iocdata   *dcp = NULL;            /* for devctl */
3717         dev_info_t              *cdip;
3718         mdi_pathinfo_t          *pip = NULL;
3719         char                    *ndi_nm;                /* NDI name */
3720         char                    *ndi_addr;              /* NDI addr */
3721         int                     is_mpxio, circ;
3722         int                     devi_entered = 0;
3723         clock_t                 end_time;

3725         ASSERT(rval != NULL);

3727         FCP_DTRACE(fcp_logq, "fcp",
3728             fcp_trace, FCP_BUF_LEVEL_8, 0,
3729             "fcp_port_ioctl(cmd=0x%x, claimed=%d)", cmd, claimed);

3731         /* if already claimed then forget it */
3732         if (claimed) {
3733                 /*
3734                  * for now, if this ioctl has already been claimed, then
3735                  * we just ignore it
3736                  */
3737                 return (retval);
3738         }

3740         /* get our port info */
3741         if ((pptr = fcp_get_port(port_handle)) == NULL) {
3742                 fcp_log(CE_WARN, NULL,
3743                     "!fcp:Invalid port handle handle in ioctl");
3744                 *rval = ENXIO;
3745                 return (retval);
3746         }
3747         is_mpxio = pptr->port_mpxio;

3749         switch (cmd) {
3750         case DEVCTL_BUS_GETSTATE:
3751         case DEVCTL_BUS_QUIESCE:
3752         case DEVCTL_BUS_UNQUIESCE:
3753         case DEVCTL_BUS_RESET:
3754         case DEVCTL_BUS_RESETALL:
```



```
3756         case DEVCTL_BUS_DEV_CREATE:
3757                 if (ndi_dc_allochdl((void *)data, &dcp) != NDI_SUCCESS) {
3758                         return (retval);
3759                 }
3760                 break;

3762         case DEVCTL_DEVICE_GETSTATE:
3763         case DEVCTL_DEVICE_OFFLINE:
3764         case DEVCTL_DEVICE_ONLINE:
3765         case DEVCTL_DEVICE_REMOVE:
3766         case DEVCTL_DEVICE_RESET:
3767                 if (ndi_dc_allochdl((void *)data, &dcp) != NDI_SUCCESS) {
3768                         return (retval);
3769                 }

3771                 ASSERT(dcp != NULL);

3773                 /* ensure we have a name and address */
3774                 if (((ndi_nm = ndi_dc_getname(dcp)) == NULL) ||
3775                     ((ndi_addr = ndi_dc_getaddr(dcp)) == NULL)) {
3776                         FCP_TRACE(fcp_logq, pptr->port_instbuf,
3777                             fcp_trace, FCP_BUF_LEVEL_2, 0,
3778                             "ioctl: can't get name (%s) or addr (%s)",
3779                             ndi_nm ? ndi_nm : "<null ptr>",
3780                             ndi_addr ? ndi_addr : "<null ptr>");
3781                         ndi_dc_freehdl(dcp);
3782                         return (retval);
3783                 }


3786                 /* get our child's DIP */
3787                 ASSERT(pptr != NULL);
3788                 if (is_mpxio) {
3789                         mdi_devi_enter(pptr->port_dip, &circ);
3790                 } else {
3791                         ndi_devi_enter(pptr->port_dip, &circ);
3792                 }
3793                 devi_entered = 1;

3795                 if ((cdip = ndi_devi_find(pptr->port_dip, ndi_nm,
3796                     ndi_addr)) == NULL) {
3797                         /* Look for virtually enumerated devices. */
3798                         pip = mdi_pi_find(pptr->port_dip, NULL, ndi_addr);
3799                         if (pip == NULL ||
3800                             ((cdip = mdi_pi_get_client(pip)) == NULL)) {
3801                                 *rval = ENXIO;
3802                                 goto out;
3803                         }
3804                 }
3805                 break;

3807         default:
3808                 *rval = ENOTTY;
3809                 return (retval);
3810         }

3812         /* this ioctl is ours -- process it */

3814         retval = FC_SUCCESS;            /* just means we claim the ioctl */

3816         /* we assume it will be a success; else we'll set error value */
3817         *rval = 0;


3820         FCP_DTRACE(fcp_logq, pptr->port_instbuf,
3821             fcp_trace, FCP_BUF_LEVEL_8, 0,
```

```
3822             "ioctl: claiming this one");

3824         /* handle ioctls now */
3825         switch (cmd) {
3826         case DEVCTL_DEVICE_GETSTATE:
3827                 ASSERT(cdip != NULL);
3828                 ASSERT(dcp != NULL);
3829                 if (ndi_dc_return_dev_state(cdip, dcp) != NDI_SUCCESS) {
3830                         *rval = EFAULT;
3831                 }
3832                 break;

3834         case DEVCTL_DEVICE_REMOVE:
3835         case DEVCTL_DEVICE_OFFLINE: {
3836                 int                     flag = 0;
3837                 int                     lcount;
3838                 int                     tcount;
3839                 struct fcp_pkt  *head = NULL;
3840                 struct fcp_lun  *plun;
3841                 child_info_t            *cip = CIP(cdip);
3842                 int                     all = 1;
3843                 struct fcp_lun  *tplun;
3844                 struct fcp_tgt  *ptgt;

3846                 ASSERT(pptr != NULL);
3847                 ASSERT(cdip != NULL);

3849                 mutex_enter(&pptr->port_mutex);
3850                 if (pip != NULL) {
3851                         cip = CIP(pip);
3852                 }
3853                 if ((plun = fcp_get_lun_from_cip(pptr, cip)) == NULL) {
3854                         mutex_exit(&pptr->port_mutex);
3855                         *rval = ENXIO;
3856                         break;
3857                 }

3859                 head = fcp_scan_commands(plun);
3860                 if (head != NULL) {
3861                         fcp_abort_commands(head, LUN_PORT);
3862                 }
3863                 lcount = pptr->port_link_cnt;
3864                 tcount = plun->lun_tgt->tgt_change_cnt;
3865                 mutex_exit(&pptr->port_mutex);

3867                 if (cmd == DEVCTL_DEVICE_REMOVE) {
3868                         flag = NDI_DEVI_REMOVE;
3869                 }

3871                 if (is_mpxio) {
3872                         mdi_devi_exit(pptr->port_dip, circ);
3873                 } else {
3874                         ndi_devi_exit(pptr->port_dip, circ);
3875                 }
3876                 devi_entered = 0;

3878                 *rval = fcp_pass_to_hp_and_wait(pptr, plun, cip,
3879                     FCP_OFFLINE, lcount, tcount, flag);

3881                 if (*rval != NDI_SUCCESS) {
3882                         *rval = (*rval == NDI_BUSY) ? EBUSY : EIO;
3883                         break;
3884                 }

3886                 fcp_update_offline_flags(plun);
```



```
3888                 ptgt = plun->lun_tgt;
3889                 mutex_enter(&ptgt->tgt_mutex);
3890                 for (tplun = ptgt->tgt_lun; tplun != NULL; tplun =
3891                     tplun->lun_next) {
3892                         mutex_enter(&tplun->lun_mutex);
3893                         if (!(tplun->lun_state & FCP_LUN_OFFLINE)) {
3894                                 all = 0;
3895                         }
3896                         mutex_exit(&tplun->lun_mutex);
3897                 }

3899                 if (all) {
3900                         ptgt->tgt_node_state = FCP_TGT_NODE_NONE;
3901                         /*
3902                          * The user is unconfiguring/offlining the device.
3903                          * If fabric and the auto configuration is set
3904                          * then make sure the user is the only one who
3905                          * can reconfigure the device.
3906                          */
3907                         if (FC_TOP_EXTERNAL(pptr->port_topology) &&
3908                             fcp_enable_auto_configuration) {
3909                                 ptgt->tgt_manual_config_only = 1;
3910                         }
3911                 }
3912                 mutex_exit(&ptgt->tgt_mutex);
3913                 break;
3914         }

3916         case DEVCTL_DEVICE_ONLINE: {
3917                 int                     lcount;
3918                 int                     tcount;
3919                 struct fcp_lun  *plun;
3920                 child_info_t            *cip = CIP(cdip);

3922                 ASSERT(cdip != NULL);
3923                 ASSERT(pptr != NULL);

3925                 mutex_enter(&pptr->port_mutex);
3926                 if (pip != NULL) {
3927                         cip = CIP(pip);
3928                 }
3929                 if ((plun = fcp_get_lun_from_cip(pptr, cip)) == NULL) {
3930                         mutex_exit(&pptr->port_mutex);
3931                         *rval = ENXIO;
3932                         break;
3933                 }
3934                 lcount = pptr->port_link_cnt;
3935                 tcount = plun->lun_tgt->tgt_change_cnt;
3936                 mutex_exit(&pptr->port_mutex);

3938                 /*
3939                  * The FCP_LUN_ONLINING flag is used in fcp_scsi_start()
3940                  * to allow the device attach to occur when the device is
3941                  * FCP_LUN_OFFLINE (so we don't reject the INQUIRY command
3942                  * from the scsi_probe()).
3943                  */
3944                 mutex_enter(&LUN_TGT->tgt_mutex);
3945                 plun->lun_state |= FCP_LUN_ONLINING;
3946                 mutex_exit(&LUN_TGT->tgt_mutex);

3948                 if (is_mpxio) {
3949                         mdi_devi_exit(pptr->port_dip, circ);
3950                 } else {
3951                         ndi_devi_exit(pptr->port_dip, circ);
3952                 }
3953                 devi_entered = 0;
```

```
3955                *rval = fcp_pass_to_hp_and_wait(pptr, plun, cip,
3956                    FCP_ONLINE, lcount, tcount, 0);

3958                if (*rval != NDI_SUCCESS) {
3959                        /* Reset the FCP_LUN_ONLINING bit */
3960                        mutex_enter(&LUN_TGT->tgt_mutex);
3961                        plun->lun_state &= ~FCP_LUN_ONLINING;
3962                        mutex_exit(&LUN_TGT->tgt_mutex);
3963                        *rval = EIO;
3964                        break;
3965                }
3966                mutex_enter(&LUN_TGT->tgt_mutex);
3967                plun->lun_state &= ~(FCP_LUN_OFFLINE | FCP_LUN_BUSY |
3968                    FCP_LUN_ONLINING);
3969                mutex_exit(&LUN_TGT->tgt_mutex);
3970                break;
3971        }

3973        case DEVCTL_BUS_DEV_CREATE: {
3974                uchar_t                 *bytes = NULL;
3975                uint_t                  nbytes;
3976                struct fcp_tgt          *ptgt = NULL;
3977                struct fcp_lun          *plun = NULL;
3978                dev_info_t              *useless_dip = NULL;

3980                *rval = ndi_dc_devi_create(dcp, pptr->port_dip,
3981                    DEVCTL_CONSTRUCT, &useless_dip);
3982                if (*rval != 0 || useless_dip == NULL) {
3983                        break;
3984                }

3986                if ((ddi_prop_lookup_byte_array(DDI_DEV_T_ANY, useless_dip,
3987                    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM, PORT_WWN_PROP, &bytes,
3988                    &nbytes) != DDI_PROP_SUCCESS) || nbytes != FC_WWN_SIZE) {
3989                        *rval = EINVAL;
3990                        (void) ndi_devi_free(useless_dip);
3991                        if (bytes != NULL) {
3992                                ddi_prop_free(bytes);
3993                        }
3994                        break;
3995                }

3997                *rval = fcp_create_on_demand(pptr, bytes);
3998                if (*rval == 0) {
3999                        mutex_enter(&pptr->port_mutex);
4000                        ptgt = fcp_lookup_target(pptr, (uchar_t *)bytes);
4001                        if (ptgt) {
4002                                /*
4003                                 * We now have a pointer to the target that
4004                                 * was created. Lets point to the first LUN on
4005                                 * this new target.
4006                                 */
4007                                mutex_enter(&ptgt->tgt_mutex);

4009                                plun = ptgt->tgt_lun;
4010                                /*
4011                                 * There may be stale/offline LUN entries on
4012                                 * this list (this is by design) and so we have
4013                                 * to make sure we point to the first online
4014                                 * LUN
4015                                 */
4016                                while (plun &&
4017                                    plun->lun_state & FCP_LUN_OFFLINE) {
4018                                        plun = plun->lun_next;
4019                                }
```



```
4021                                        mutex_exit(&ptgt->tgt_mutex);
4022                                }
4023                                mutex_exit(&pptr->port_mutex);
4024                        }

4026                        if (*rval == 0 && ptgt && plun) {
4027                                mutex_enter(&plun->lun_mutex);
4028                                /*
4029                                 * Allow up to fcp_lun_ready_retry seconds to
4030                                 * configure all the luns behind the target.
4031                                 *
4032                                 * The intent here is to allow targets with long
4033                                 * reboot/reset-recovery times to become available
4034                                 * while limiting the maximum wait time for an
4035                                 * unresponsive target.
4036                                 */
4037                                end_time = ddi_get_lbolt() +
4038                                    SEC_TO_TICK(fcp_lun_ready_retry);

4040                                while (ddi_get_lbolt() < end_time) {
4041                                        retval = FC_SUCCESS;

4043                                        /*
4044                                         * The new ndi interfaces for on-demand creation
4045                                         * are inflexible, Do some more work to pass on
4046                                         * a path name of some LUN (design is broken !)
4047                                         */
4048                                        if (plun->lun_cip) {
4049                                                if (plun->lun_mpxio == 0) {
4050                                                        cdip = DIP(plun->lun_cip);
4051                                                } else {
4052                                                        cdip = mdi_pi_get_client(
4053                                                            PIP(plun->lun_cip));
4054                                                }
4055                                                if (cdip == NULL) {
4056                                                        *rval = ENXIO;
4057                                                        break;
4058                                                }

4060                                                if (!i_ddi_devi_attached(cdip)) {
4061                                                        mutex_exit(&plun->lun_mutex);
4062                                                        delay(drv_sectohz(1));
4062                                                        delay(drv_usectohz(1000000));
4063                                                        mutex_enter(&plun->lun_mutex);
4064                                                } else {
4065                                                        /*
4066                                                         * This Lun is ready, lets
4067                                                         * check the next one.
4068                                                         */
4069                                                        mutex_exit(&plun->lun_mutex);
4070                                                        plun = plun->lun_next;
4071                                                        while (plun && (plun->lun_state
4072                                                            & FCP_LUN_OFFLINE)) {
4073                                                                plun = plun->lun_next;
4074                                                        }
4075                                                        if (!plun) {
4076                                                                break;
4077                                                        }
4078                                                        mutex_enter(&plun->lun_mutex);
4079                                                }
4080                                        } else {
4081                                                /*
4082                                                 * lun_cip field for a valid lun
4083                                                 * should never be NULL. Fail the
4084                                                 * command.
```

```
4085                                     */
4086                                    *rval = ENXIO;
4087                                    break;
4088                            }
4089                    }
4090                    if (plun) {
4091                            mutex_exit(&plun->lun_mutex);
4092                    } else {
4093                            char devnm[MAXNAMELEN];
4094                            int nmlen;

4096                            nmlen = snprintf(devnm, MAXNAMELEN, "%s@%s",
4097                                ddi_node_name(cdip),
4098                                ddi_get_name_addr(cdip));

4100                            if (copyout(&devnm, dcp->cpyout_buf, nmlen) !=
4101                                0) {
4102                                    *rval = EFAULT;
4103                            }
4104                    }
4105            } else {
4106                    int     i;
4107                    char    buf[25];

4109                    for (i = 0; i < FC_WWN_SIZE; i++) {
4110                            (void) sprintf(&buf[i << 1], "%02x", bytes[i]);
4111                    }

4113                    fcp_log(CE_WARN, pptr->port_dip,
4114                        "!Failed to create nodes for pwwn=%s; error=%x",
4115                        buf, *rval);
4116            }

4118            (void) ndi_devi_free(useless_dip);
4119            ddi_prop_free(bytes);
4120            break;
4121    }

4123    case DEVCTL_DEVICE_RESET: {
4124            struct fcp_lun          *plun;
4125            child_info_t            *cip = CIP(cdip);

4127            ASSERT(cdip != NULL);
4128            ASSERT(pptr != NULL);
4129            mutex_enter(&pptr->port_mutex);
4130            if (pip != NULL) {
4131                    cip = CIP(pip);
4132            }
4133            if ((plun = fcp_get_lun_from_cip(pptr, cip)) == NULL) {
4134                    mutex_exit(&pptr->port_mutex);
4135                    *rval = ENXIO;
4136                    break;
4137            }
4138            mutex_exit(&pptr->port_mutex);

4140            mutex_enter(&plun->lun_tgt->tgt_mutex);
4141            if (!(plun->lun_state & FCP_SCSI_LUN_TGT_INIT)) {
4142                    mutex_exit(&plun->lun_tgt->tgt_mutex);

4144                    *rval = ENXIO;
4145                    break;
4146            }

4148            if (plun->lun_sd == NULL) {
4149                    mutex_exit(&plun->lun_tgt->tgt_mutex);
```



```
4151                            *rval = ENXIO;
4152                            break;
4153                    }
4154                    mutex_exit(&plun->lun_tgt->tgt_mutex);

4156                    /*
4157                     * set up ap so that fcp_reset can figure out
4158                     * which target to reset
4159                     */
4160                    if (fcp_scsi_reset(&plun->lun_sd->sd_address,
4161                        RESET_TARGET) == FALSE) {
4162                            *rval = EIO;
4163                    }
4164                    break;
4165            }

4167            case DEVCTL_BUS_GETSTATE:
4168                    ASSERT(dcp != NULL);
4169                    ASSERT(pptr != NULL);
4170                    ASSERT(pptr->port_dip != NULL);
4171                    if (ndi_dc_return_bus_state(pptr->port_dip, dcp) !=
4172                        NDI_SUCCESS) {
4173                            *rval = EFAULT;
4174                    }
4175                    break;

4177            case DEVCTL_BUS_QUIESCE:
4178            case DEVCTL_BUS_UNQUIESCE:
4179                    *rval = ENOTSUP;
4180                    break;

4182            case DEVCTL_BUS_RESET:
4183            case DEVCTL_BUS_RESETALL:
4184                    ASSERT(pptr != NULL);
4185                    (void) fcp_linkreset(pptr, NULL,  KM_SLEEP);
4186                    break;

4188            default:
4189                    ASSERT(dcp != NULL);
4190                    *rval = ENOTTY;
4191                    break;
4192            }

4194            /* all done -- clean up and return */
4195    out:    if (devi_entered) {
4196                    if (is_mpxio) {
4197                            mdi_devi_exit(pptr->port_dip, circ);
4198                    } else {
4199                            ndi_devi_exit(pptr->port_dip, circ);
4200                    }
4201            }

4203            if (dcp != NULL) {
4204                    ndi_dc_freehdl(dcp);
4205            }

4207            return (retval);
4208    }
```

_____***unchanged_portion_omitted_***

```
4272 /*
4273  *     Function: fcp_statec_callback
4274  *
4275  *  Description: The purpose of this function is to handle a port state change.
4276  *               It is called from fp/fctl and, in a few instances, internally.
4277  *
```

```
 *     Argument: ulph           fp/fctl port handle
 *               port_handle    fcp_port structure
 *               port_state     Physical state of the port
 *               port_top       Topology
 *               *devlist       Pointer to the first entry of a table
 *                              containing the remote ports that can be
 *                              reached.
 *               dev_cnt        Number of entries pointed by devlist.
 *               port_sid       Port ID of the local port.
 *
 * Return Value: None
 */
/*ARGSUSED*/
static void
fcp_statec_callback(opaque_t ulph, opaque_t port_handle,
    uint32_t port_state, uint32_t port_top, fc_portmap_t *devlist,
    uint32_t dev_cnt, uint32_t port_sid)
{
        uint32_t                link_count;
        int                     map_len = 0;
        struct fcp_port *pptr;
        fcp_map_tag_t           *map_tag = NULL;

        if ((pptr = fcp_get_port(port_handle)) == NULL) {
                fcp_log(CE_WARN, NULL, "!Invalid port handle in callback");
                return;                 /* nothing to work with! */
        }

        FCP_TRACE(fcp_logq, pptr->port_instbuf,
            fcp_trace, FCP_BUF_LEVEL_2, 0,
            "fcp_statec_callback: port state/dev_cnt/top ="
            "%d/%d/%d", FC_PORT_STATE_MASK(port_state),
            dev_cnt, port_top);

        mutex_enter(&pptr->port_mutex);

        /*
         * If a thread is in detach, don't do anything.
         */
        if (pptr->port_state & (FCP_STATE_DETACHING |
            FCP_STATE_SUSPENDED | FCP_STATE_POWER_DOWN)) {
                mutex_exit(&pptr->port_mutex);
                return;
        }

        /*
         * First thing we do is set the FCP_STATE_IN_CB_DEVC flag so that if
         * init_pkt is called, it knows whether or not the target's status
         * (or pd) might be changing.
         */

        if (FC_PORT_STATE_MASK(port_state) == FC_STATE_DEVICE_CHANGE) {
                pptr->port_state |= FCP_STATE_IN_CB_DEVC;
        }

        /*
         * the transport doesn't allocate or probe unless being
         * asked to by either the applications or ULPs
         *
         * in cases where the port is OFFLINE at the time of port
         * attach callback and the link comes ONLINE later, for
         * easier automatic node creation (i.e. without you having to
         * go out and run the utility to perform LOGINs) the
         * following conditional is helpful
         */
        pptr->port_phys_state = port_state;
```



```
        if (dev_cnt) {
                mutex_exit(&pptr->port_mutex);

                map_len = sizeof (*map_tag) * dev_cnt;
                map_tag = kmem_alloc(map_len, KM_NOSLEEP);
                if (map_tag == NULL) {
                        fcp_log(CE_WARN, pptr->port_dip,
                            "!fcp%d: failed to allocate for map tags; "
                            " state change will not be processed",
                            pptr->port_instance);

                        mutex_enter(&pptr->port_mutex);
                        pptr->port_state &= ~FCP_STATE_IN_CB_DEVC;
                        mutex_exit(&pptr->port_mutex);

                        return;
                }

                mutex_enter(&pptr->port_mutex);
        }

        if (pptr->port_id != port_sid) {
                FCP_TRACE(fcp_logq, pptr->port_instbuf,
                    fcp_trace, FCP_BUF_LEVEL_3, 0,
                    "fcp: Port S_ID=0x%x => 0x%x", pptr->port_id,
                    port_sid);
                /*
                 * The local port changed ID. It is the first time a port ID
                 * is assigned or something drastic happened.  We might have
                 * been unplugged and replugged on another loop or fabric port
                 * or somebody grabbed the AL_PA we had or somebody rezoned
                 * the fabric we were plugged into.
                 */
                pptr->port_id = port_sid;
        }

        switch (FC_PORT_STATE_MASK(port_state)) {
        case FC_STATE_OFFLINE:
        case FC_STATE_RESET_REQUESTED:
                /*
                 * link has gone from online to offline -- just update the
                 * state of this port to BUSY and MARKed to go offline
                 */
                FCP_TRACE(fcp_logq, pptr->port_instbuf,
                    fcp_trace, FCP_BUF_LEVEL_3, 0,
                    "link went offline");
                if ((pptr->port_state & FCP_STATE_OFFLINE) && dev_cnt) {
                        /*
                         * We were offline a while ago and this one
                         * seems to indicate that the loop has gone
                         * dead forever.
                         */
                        pptr->port_tmp_cnt += dev_cnt;
                        pptr->port_state &= ~FCP_STATE_OFFLINE;
                        pptr->port_state |= FCP_STATE_INIT;
                        link_count = pptr->port_link_cnt;
                        fcp_handle_devices(pptr, devlist, dev_cnt,
                            link_count, map_tag, FCP_CAUSE_LINK_DOWN);
                } else {
                        pptr->port_link_cnt++;
                        ASSERT(!(pptr->port_state & FCP_STATE_SUSPENDED));
                        fcp_update_state(pptr, (FCP_LUN_BUSY |
                            FCP_LUN_MARK), FCP_CAUSE_LINK_DOWN);
                        if (pptr->port_mpxio) {
                                fcp_update_mpxio_path_verifybusy(pptr);
```

```
4410                        }
4411                        pptr->port_state |= FCP_STATE_OFFLINE;
4412                        pptr->port_state &=
4413                            ~(FCP_STATE_ONLINING | FCP_STATE_ONLINE);
4414                        pptr->port_tmp_cnt = 0;
4415                }
4416                mutex_exit(&pptr->port_mutex);
4417                break;

4419        case FC_STATE_ONLINE:
4420        case FC_STATE_LIP:
4421        case FC_STATE_LIP_LBIT_SET:
4422                /*
4423                 * link has gone from offline to online
4424                 */
4425                FCP_TRACE(fcp_logq, pptr->port_instbuf,
4426                    fcp_trace, FCP_BUF_LEVEL_3, 0,
4427                    "link went online");

4429                pptr->port_link_cnt++;

4431                while (pptr->port_ipkt_cnt) {
4432                        mutex_exit(&pptr->port_mutex);
4433                        delay(drv_sectohz(1));
4433                        delay(drv_usectohz(1000000));
4434                        mutex_enter(&pptr->port_mutex);
4435                }

4437                pptr->port_topology = port_top;

4439                /*
4440                 * The state of the targets and luns accessible through this
4441                 * port is updated.
4442                 */
4443                fcp_update_state(pptr, FCP_LUN_BUSY | FCP_LUN_MARK,
4444                    FCP_CAUSE_LINK_CHANGE);

4446                pptr->port_state &= ~(FCP_STATE_INIT | FCP_STATE_OFFLINE);
4447                pptr->port_state |= FCP_STATE_ONLINING;
4448                pptr->port_tmp_cnt = dev_cnt;
4449                link_count = pptr->port_link_cnt;

4451                pptr->port_deadline = fcp_watchdog_time +
4452                    FCP_ICMD_DEADLINE;

4454                if (!dev_cnt) {
4455                        /*
4456                         * We go directly to the online state if no remote
4457                         * ports were discovered.
4458                         */
4459                        FCP_TRACE(fcp_logq, pptr->port_instbuf,
4460                            fcp_trace, FCP_BUF_LEVEL_3, 0,
4461                            "No remote ports discovered");

4463                        pptr->port_state &= ~FCP_STATE_ONLINING;
4464                        pptr->port_state |= FCP_STATE_ONLINE;
4465                }

4467                switch (port_top) {
4468                case FC_TOP_FABRIC:
4469                case FC_TOP_PUBLIC_LOOP:
4470                case FC_TOP_PRIVATE_LOOP:
4471                case FC_TOP_PT_PT:

4473                        if (pptr->port_state & FCP_STATE_NS_REG_FAILED) {
4474                                fcp_retry_ns_registry(pptr, port_sid);
```



```
4475                        }

4477                        fcp_handle_devices(pptr, devlist, dev_cnt, link_count,
4478                            map_tag, FCP_CAUSE_LINK_CHANGE);
4479                        break;

4481                default:
4482                        /*
4483                         * We got here because we were provided with an unknown
4484                         * topology.
4485                         */
4486                        if (pptr->port_state & FCP_STATE_NS_REG_FAILED) {
4487                                pptr->port_state &= ~FCP_STATE_NS_REG_FAILED;
4488                        }

4490                        pptr->port_tmp_cnt -= dev_cnt;
4491                        fcp_log(CE_WARN, pptr->port_dip,
4492                            "!unknown/unsupported topology (0x%x)", port_top);
4493                        break;
4494                }
4495                FCP_TRACE(fcp_logq, pptr->port_instbuf,
4496                    fcp_trace, FCP_BUF_LEVEL_3, 0,
4497                    "Notify ssd of the reset to reinstate the reservations");

4499                scsi_hba_reset_notify_callback(&pptr->port_mutex,
4500                    &pptr->port_reset_notify_listf);

4502                mutex_exit(&pptr->port_mutex);

4504                break;

4506        case FC_STATE_RESET:
4507                ASSERT(pptr->port_state & FCP_STATE_OFFLINE);
4508                FCP_TRACE(fcp_logq, pptr->port_instbuf,
4509                    fcp_trace, FCP_BUF_LEVEL_3, 0,
4510                    "RESET state, waiting for Offline/Online state_cb");
4511                mutex_exit(&pptr->port_mutex);
4512                break;

4514        case FC_STATE_DEVICE_CHANGE:
4515                /*
4516                 * We come here when an application has requested
4517                 * Dynamic node creation/deletion in Fabric connectivity.
4518                 */
4519                if (pptr->port_state & (FCP_STATE_OFFLINE |
4520                    FCP_STATE_INIT)) {
4521                        /*
4522                         * This case can happen when the FCTL is in the
4523                         * process of giving us on online and the host on
4524                         * the other side issues a PLOGI/PLOGO. Ideally
4525                         * the state changes should be serialized unless
4526                         * they are opposite (online-offline).
4527                         * The transport will give us a final state change
4528                         * so we can ignore this for the time being.
4529                         */
4530                        pptr->port_state &= ~FCP_STATE_IN_CB_DEVC;
4531                        mutex_exit(&pptr->port_mutex);
4532                        break;
4533                }

4535                if (pptr->port_state & FCP_STATE_NS_REG_FAILED) {
4536                        fcp_retry_ns_registry(pptr, port_sid);
4537                }

4539                /*
4540                 * Extend the deadline under steady state conditions
```

```
4541                 * to provide more time for the device-change-commands
4542                 */
4543                if (!pptr->port_ipkt_cnt) {
4544                        pptr->port_deadline = fcp_watchdog_time +
4545                            FCP_ICMD_DEADLINE;
4546                }

4548                /*
4549                 * There is another race condition here, where if we were
4550                 * in ONLINEING state and a devices in the map logs out,
4551                 * fp will give another state change as DEVICE_CHANGE
4552                 * and OLD. This will result in that target being offlined.
4553                 * The pd_handle is freed. If from the first statec callback
4554                 * we were going to fire a PLOGI/PRLI, the system will
4555                 * panic in fc_ulp_transport with invalid pd_handle.
4556                 * The fix is to check for the link_cnt before issuing
4557                 * any command down.
4558                 */
4559                fcp_update_targets(pptr, devlist, dev_cnt,
4560                    FCP_LUN_BUSY | FCP_LUN_MARK, FCP_CAUSE_TGT_CHANGE);

4562                link_count = pptr->port_link_cnt;

4564                fcp_handle_devices(pptr, devlist, dev_cnt,
4565                    link_count, map_tag, FCP_CAUSE_TGT_CHANGE);

4567                pptr->port_state &= ~FCP_STATE_IN_CB_DEVC;

4569                mutex_exit(&pptr->port_mutex);
4570                break;

4572        case FC_STATE_TARGET_PORT_RESET:
4573                if (pptr->port_state & FCP_STATE_NS_REG_FAILED) {
4574                        fcp_retry_ns_registry(pptr, port_sid);
4575                }

4577                /* Do nothing else */
4578                mutex_exit(&pptr->port_mutex);
4579                break;

4581        default:
4582                fcp_log(CE_WARN, pptr->port_dip,
4583                    "!Invalid state change=0x%x", port_state);
4584                mutex_exit(&pptr->port_mutex);
4585                break;
4586        }

4588        if (map_tag) {
4589                kmem_free(map_tag, map_len);
4590        }
4591 }

4593 /*
4594  *     Function: fcp_handle_devices
4595  *
4596  *  Description: This function updates the devices currently known by
4597  *               walking the list provided by the caller.  The list passed
4598  *               by the caller is supposed to be the list of reachable
4599  *               devices.
4600  *
4601  *     Argument: *pptr          Fcp port structure.
4602  *               *devlist       Pointer to the first entry of a table
4603  *                              containing the remote ports that can be
4604  *                              reached.
4605  *               dev_cnt        Number of entries pointed by devlist.
4606  *               link_cnt       Link state count.
```



```
4607  *               *map_tag       Array of fcp_map_tag_t structures.
4608  *               cause          What caused this function to be called.
4609  *
4610  * Return Value: None
4611  *
4612  *        Notes: The pptr->port_mutex must be held.
4613  */
4614 static void
4615 fcp_handle_devices(struct fcp_port *pptr, fc_portmap_t devlist[],
4616     uint32_t dev_cnt, int link_cnt, fcp_map_tag_t *map_tag, int cause)
4617 {
4618        int                     i;
4619        int                     check_finish_init = 0;
4620        fc_portmap_t            *map_entry;
4621        struct fcp_tgt  *ptgt = NULL;

4623        FCP_TRACE(fcp_logq, pptr->port_instbuf,
4624            fcp_trace, FCP_BUF_LEVEL_3, 0,
4625            "fcp_handle_devices: called for %d dev(s)", dev_cnt);

4627        if (dev_cnt) {
4628                ASSERT(map_tag != NULL);
4629        }

4631        /*
4632         * The following code goes through the list of remote ports that are
4633         * accessible through this (pptr) local port (The list walked is the
4634         * one provided by the caller which is the list of the remote ports
4635         * currently reachable).  It checks if any of them was already
4636         * known by looking for the corresponding target structure based on
4637         * the world wide name.  If a target is part of the list it is tagged
4638         * (ptgt->tgt_aux_state = FCP_TGT_TAGGED).
4639         *
4640         * Old comment
4641         * -----------
4642         * Before we drop port mutex; we MUST get the tags updated; This
4643         * two step process is somewhat slow, but more reliable.
4644         */
4645        for (i = 0; (i < dev_cnt) && (pptr->port_link_cnt == link_cnt); i++) {
4646                map_entry = &(devlist[i]);

4648                /*
4649                 * get ptr to this map entry in our port's
4650                 * list (if any)
4651                 */
4652                ptgt = fcp_lookup_target(pptr,
4653                    (uchar_t *)&(map_entry->map_pwwn));

4655                if (ptgt) {
4656                        map_tag[i] = ptgt->tgt_change_cnt;
4657                        if (cause == FCP_CAUSE_LINK_CHANGE) {
4658                                ptgt->tgt_aux_state = FCP_TGT_TAGGED;
4659                        }
4660                }
4661        }

4663        /*
4664         * At this point we know which devices of the new list were already
4665         * known (The field tgt_aux_state of the target structure has been
4666         * set to FCP_TGT_TAGGED).
4667         *
4668         * The following code goes through the list of targets currently known
4669         * by the local port (the list is actually a hashing table).  If a
4670         * target is found and is not tagged, it means the target cannot
4671         * be reached anymore through the local port (pptr).  It is offlined.
4672         * The offlining only occurs if the cause is FCP_CAUSE_LINK_CHANGE.
```

```
4673          */
4674         for (i = 0; i < FCP_NUM_HASH; i++) {
4675                 for (ptgt = pptr->port_tgt_hash_table[i]; ptgt != NULL;
4676                     ptgt = ptgt->tgt_next) {
4677                         mutex_enter(&ptgt->tgt_mutex);
4678                         if ((ptgt->tgt_aux_state != FCP_TGT_TAGGED) &&
4679                             (cause == FCP_CAUSE_LINK_CHANGE) &&
4680                             !(ptgt->tgt_state & FCP_TGT_OFFLINE)) {
4681                                 fcp_offline_target_now(pptr, ptgt,
4682                                     link_cnt, ptgt->tgt_change_cnt, 0);
4683                         }
4684                         mutex_exit(&ptgt->tgt_mutex);
4685                 }
4686         }

4688         /*
4689          * At this point, the devices that were known but cannot be reached
4690          * anymore, have most likely been offlined.
4691          *
4692          * The following section of code seems to go through the list of
4693          * remote ports that can now be reached.  For every single one it
4694          * checks if it is already known or if it is a new port.
4695          */
4696         for (i = 0; (i < dev_cnt) && (pptr->port_link_cnt == link_cnt); i++) {

4698                 if (check_finish_init) {
4699                         ASSERT(i > 0);
4700                         (void) fcp_call_finish_init_held(pptr, ptgt, link_cnt,
4701                             map_tag[i - 1], cause);
4702                         check_finish_init = 0;
4703                 }

4705                 /* get a pointer to this map entry */
4706                 map_entry = &(devlist[i]);

4708                 /*
4709                  * Check for the duplicate map entry flag. If we have marked
4710                  * this entry as a duplicate we skip it since the correct
4711                  * (perhaps even same) state change will be encountered
4712                  * later in the list.
4713                  */
4714                 if (map_entry->map_flags & PORT_DEVICE_DUPLICATE_MAP_ENTRY) {
4715                         continue;
4716                 }

4718                 /* get ptr to this map entry in our port's list (if any) */
4719                 ptgt = fcp_lookup_target(pptr,
4720                     (uchar_t *)&(map_entry->map_pwwn));

4722                 if (ptgt) {
4723                         /*
4724                          * This device was already known.  The field
4725                          * tgt_aux_state is reset (was probably set to
4726                          * FCP_TGT_TAGGED previously in this routine).
4727                          */
4728                         ptgt->tgt_aux_state = 0;
4729                         FCP_TRACE(fcp_logq, pptr->port_instbuf,
4730                             fcp_trace, FCP_BUF_LEVEL_3, 0,
4731                             "handle_devices: map did/state/type/flags = "
4732                             "0x%x/0x%x/0x%x/0x%x, tgt_d_id=0x%x, "
4733                             "tgt_state=%d",
4734                             map_entry->map_did.port_id, map_entry->map_state,
4735                             map_entry->map_type, map_entry->map_flags,
4736                             ptgt->tgt_d_id, ptgt->tgt_state);
4737                 }
```



```
4739                 if (map_entry->map_type == PORT_DEVICE_OLD ||
4740                     map_entry->map_type == PORT_DEVICE_NEW ||
4741                     map_entry->map_type == PORT_DEVICE_REPORTLUN_CHANGED ||
4742                     map_entry->map_type == PORT_DEVICE_CHANGED) {
4743                         FCP_TRACE(fcp_logq, pptr->port_instbuf,
4744                             fcp_trace, FCP_BUF_LEVEL_2, 0,
4745                             "map_type=%x, did = %x",
4746                             map_entry->map_type,
4747                             map_entry->map_did.port_id);
4748                 }

4750                 switch (map_entry->map_type) {
4751                 case PORT_DEVICE_NOCHANGE:
4752                 case PORT_DEVICE_USER_CREATE:
4753                 case PORT_DEVICE_USER_LOGIN:
4754                 case PORT_DEVICE_NEW:
4755                 case PORT_DEVICE_REPORTLUN_CHANGED:
4756                         FCP_TGT_TRACE(ptgt, map_tag[i], FCP_TGT_TRACE_1);

4758                         if (fcp_handle_mapflags(pptr, ptgt, map_entry,
4759                             link_cnt, (ptgt) ? map_tag[i] : 0,
4760                             cause) == TRUE) {

4762                                 FCP_TGT_TRACE(ptgt, map_tag[i],
4763                                     FCP_TGT_TRACE_2);
4764                                 check_finish_init++;
4765                         }
4766                         break;

4768                 case PORT_DEVICE_OLD:
4769                         if (ptgt != NULL) {
4770                                 FCP_TGT_TRACE(ptgt, map_tag[i],
4771                                     FCP_TGT_TRACE_3);

4773                                 mutex_enter(&ptgt->tgt_mutex);
4774                                 if (!(ptgt->tgt_state & FCP_TGT_OFFLINE)) {
4775                                         /*
4776                                          * Must do an in-line wait for I/Os
4777                                          * to get drained
4778                                          */
4779                                         mutex_exit(&ptgt->tgt_mutex);
4780                                         mutex_exit(&pptr->port_mutex);

4782                                         mutex_enter(&ptgt->tgt_mutex);
4783                                         while (ptgt->tgt_ipkt_cnt ||
4784                                             fcp_outstanding_lun_cmds(ptgt)
4785                                             == FC_SUCCESS) {
4786                                                 mutex_exit(&ptgt->tgt_mutex);
4787                                                 delay(drv_sectohz(1));
4787                                                 delay(drv_usectohz(1000000));
4788                                                 mutex_enter(&ptgt->tgt_mutex);
4789                                         }
4790                                         mutex_exit(&ptgt->tgt_mutex);

4792                                         mutex_enter(&pptr->port_mutex);
4793                                         mutex_enter(&ptgt->tgt_mutex);

4795                                         (void) fcp_offline_target(pptr, ptgt,
4796                                             link_cnt, map_tag[i], 0, 0);
4797                                 }
4798                                 mutex_exit(&ptgt->tgt_mutex);
4799                         }
4800                         check_finish_init++;
4801                         break;

4803                 case PORT_DEVICE_USER_DELETE:
```

```
4804                 case PORT_DEVICE_USER_LOGOUT:
4805                         if (ptgt != NULL) {
4806                                 FCP_TGT_TRACE(ptgt, map_tag[i],
4807                                     FCP_TGT_TRACE_4);

4809                                 mutex_enter(&ptgt->tgt_mutex);
4810                                 if (!(ptgt->tgt_state & FCP_TGT_OFFLINE)) {
4811                                         (void) fcp_offline_target(pptr, ptgt,
4812                                             link_cnt, map_tag[i], 1, 0);
4813                                 }
4814                                 mutex_exit(&ptgt->tgt_mutex);
4815                         }
4816                         check_finish_init++;
4817                         break;

4819                 case PORT_DEVICE_CHANGED:
4820                         if (ptgt != NULL) {
4821                                 FCP_TGT_TRACE(ptgt, map_tag[i],
4822                                     FCP_TGT_TRACE_5);

4824                                 if (fcp_device_changed(pptr, ptgt,
4825                                     map_entry, link_cnt, map_tag[i],
4826                                     cause) == TRUE) {
4827                                         check_finish_init++;
4828                                 }
4829                         } else {
4830                                 if (fcp_handle_mapflags(pptr, ptgt,
4831                                     map_entry, link_cnt, 0, cause) == TRUE) {
4832                                         check_finish_init++;
4833                                 }
4834                         }
4835                         break;

4837                 default:
4838                         fcp_log(CE_WARN, pptr->port_dip,
4839                             "!Invalid map_type=0x%x", map_entry->map_type);
4840                         check_finish_init++;
4841                         break;
4842                 }
4843         }

4845         if (check_finish_init && pptr->port_link_cnt == link_cnt) {
4846                 ASSERT(i > 0);
4847                 (void) fcp_call_finish_init_held(pptr, ptgt, link_cnt,
4848                     map_tag[i-1], cause);
4849         } else if (dev_cnt == 0 && pptr->port_link_cnt == link_cnt) {
4850                 fcp_offline_all(pptr, link_cnt, cause);
4851         }
4852 }
```

*_____unchanged_portion_omitted_*

```
9750 /*
9751  *     Function: fcp_handle_port_attach
9752  *
9753  *  Description: This function is called from fcp_port_attach() to attach a
9754  *               new port. This routine does the following:
9755  *
9756  *               1) Allocates an fcp_port structure and initializes it.
9757  *               2) Tries to register the new FC-4 (FCP) capablity with the name
9758  *                  server.
9759  *               3) Kicks off the enumeration of the targets/luns visible
9760  *                  through this new port.  That is done by calling
9761  *                  fcp_statec_callback() if the port is online.
9762  *
9763  *     Argument: ulph          fp/fctl port handle.
9764  *               *pinfo        Port information.
```



```
9765  *               s_id          Port ID.
9766  *               instance      Device instance number for the local port
9767  *                             (returned by ddi_get_instance()).
9768  *
9769  * Return Value: DDI_SUCCESS
9770  *               DDI_FAILURE
9771  *
9772  *      Context: User and Kernel context.
9773  */
9774 /*ARGSUSED*/
9775 int
9776 fcp_handle_port_attach(opaque_t ulph, fc_ulp_port_info_t *pinfo,
9777     uint32_t s_id, int instance)
9778 {
9779         int                     res = DDI_FAILURE;
9780         scsi_hba_tran_t         *tran;
9781         int                     mutex_initted = FALSE;
9782         int                     hba_attached = FALSE;
9783         int                     soft_state_linked = FALSE;
9784         int                     event_bind = FALSE;
9785         struct fcp_port         *pptr;
9786         fc_portmap_t            *tmp_list = NULL;
9787         uint32_t                max_cnt, alloc_cnt;
9788         uchar_t                 *boot_wwn = NULL;
9789         uint_t                  nbytes;
9790         int                     manual_cfg;

9792         /*
9793          * this port instance attaching for the first time (or after
9794          * being detached before)
9795          */
9796         FCP_TRACE(fcp_logq, "fcp", fcp_trace,
9797             FCP_BUF_LEVEL_3, 0, "port attach: for port %d", instance);

9799         if (ddi_soft_state_zalloc(fcp_softstate, instance) != DDI_SUCCESS) {
9800                 cmn_err(CE_WARN, "fcp: Softstate struct alloc failed"
9801                     "parent dip: %p; instance: %d", (void *)pinfo->port_dip,
9802                     instance);
9803                 return (res);
9804         }

9806         if ((pptr = ddi_get_soft_state(fcp_softstate, instance)) == NULL) {
9807                 /* this shouldn't happen */
9808                 ddi_soft_state_free(fcp_softstate, instance);
9809                 cmn_err(CE_WARN, "fcp: bad soft state");
9810                 return (res);
9811         }

9813         (void) sprintf(pptr->port_instbuf, "fcp(%d)", instance);

9815         /*
9816          * Make a copy of ulp_port_info as fctl allocates
9817          * a temp struct.
9818          */
9819         (void) fcp_cp_pinfo(pptr, pinfo);

9821         /*
9822          * Check for manual_configuration_only property.
9823          * Enable manual configurtion if the property is
9824          * set to 1, otherwise disable manual configuration.
9825          */
9826         if ((manual_cfg = ddi_prop_get_int(DDI_DEV_T_ANY, pptr->port_dip,
9827             DDI_PROP_NOTPROM | DDI_PROP_DONTPASS,
9828             MANUAL_CFG_ONLY,
9829             -1)) != -1) {
9830                 if (manual_cfg == 1) {
```

```
9831                        char    *pathname;
9832                        pathname = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
9833                        (void) ddi_pathname(pptr->port_dip, pathname);
9834                        cmn_err(CE_NOTE,
9835                            "%s (%s%d) %s is enabled via %s.conf.",
9836                            pathname,
9837                            ddi_driver_name(pptr->port_dip),
9838                            ddi_get_instance(pptr->port_dip),
9839                            MANUAL_CFG_ONLY,
9840                            ddi_driver_name(pptr->port_dip));
9841                        fcp_enable_auto_configuration = 0;
9842                        kmem_free(pathname, MAXPATHLEN);
9843                }
9844        }
9845        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(pptr->port_link_cnt));
9846        pptr->port_link_cnt = 1;
9847        _NOTE(NOW_VISIBLE_TO_OTHER_THREADS(pptr->port_link_cnt));
9848        pptr->port_id = s_id;
9849        pptr->port_instance = instance;
9850        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(pptr->port_state));
9851        pptr->port_state = FCP_STATE_INIT;
9852        if (pinfo->port_acc_attr == NULL) {
9853                /*
9854                 * The corresponding FCA doesn't support DMA at all
9855                 */
9856                pptr->port_state |= FCP_STATE_FCA_IS_NODMA;
9857        }

9859        _NOTE(NOW_VISIBLE_TO_OTHER_THREADS(pptr->port_state));

9861        if (!(pptr->port_state & FCP_STATE_FCA_IS_NODMA)) {
9862                /*
9863                 * If FCA supports DMA in SCSI data phase, we need preallocate
9864                 * dma cookie, so stash the cookie size
9865                 */
9866                pptr->port_dmacookie_sz = sizeof (ddi_dma_cookie_t) *
9867                    pptr->port_data_dma_attr.dma_attr_sgllen;
9868        }

9870        /*
9871         * The two mutexes of fcp_port are initialized.  The variable
9872         * mutex_initted is incremented to remember that fact.  That variable
9873         * is checked when the routine fails and the mutexes have to be
9874         * destroyed.
9875         */
9876        mutex_init(&pptr->port_mutex, NULL, MUTEX_DRIVER, NULL);
9877        mutex_init(&pptr->port_pkt_mutex, NULL, MUTEX_DRIVER, NULL);
9878        mutex_initted++;

9880        /*
9881         * The SCSI tran structure is allocate and initialized now.
9882         */
9883        if ((tran = scsi_hba_tran_alloc(pptr->port_dip, 0)) == NULL) {
9884                fcp_log(CE_WARN, pptr->port_dip,
9885                    "!fcp%d: scsi_hba_tran_alloc failed", instance);
9886                goto fail;
9887        }

9889        /* link in the transport structure then fill it in */
9890        pptr->port_tran = tran;
9891        tran->tran_hba_private          = pptr;
9892        tran->tran_tgt_init             = fcp_scsi_tgt_init;
9893        tran->tran_tgt_probe            = NULL;
9894        tran->tran_tgt_free             = fcp_scsi_tgt_free;
9895        tran->tran_start                = fcp_scsi_start;
9896        tran->tran_reset                = fcp_scsi_reset;
```



```
9897        tran->tran_abort                = fcp_scsi_abort;
9898        tran->tran_getcap               = fcp_scsi_getcap;
9899        tran->tran_setcap               = fcp_scsi_setcap;
9900        tran->tran_init_pkt             = NULL;
9901        tran->tran_destroy_pkt          = NULL;
9902        tran->tran_dmafree              = NULL;
9903        tran->tran_sync_pkt             = NULL;
9904        tran->tran_reset_notify         = fcp_scsi_reset_notify;
9905        tran->tran_get_bus_addr         = fcp_scsi_get_bus_addr;
9906        tran->tran_get_name             = fcp_scsi_get_name;
9907        tran->tran_clear_aca            = NULL;
9908        tran->tran_clear_task_set       = NULL;
9909        tran->tran_terminate_task       = NULL;
9910        tran->tran_get_eventcookie      = fcp_scsi_bus_get_eventcookie;
9911        tran->tran_add_eventcall        = fcp_scsi_bus_add_eventcall;
9912        tran->tran_remove_eventcall     = fcp_scsi_bus_remove_eventcall;
9913        tran->tran_post_event           = fcp_scsi_bus_post_event;
9914        tran->tran_quiesce              = NULL;
9915        tran->tran_unquiesce            = NULL;
9916        tran->tran_bus_reset            = NULL;
9917        tran->tran_bus_config           = fcp_scsi_bus_config;
9918        tran->tran_bus_unconfig         = fcp_scsi_bus_unconfig;
9919        tran->tran_bus_power            = NULL;
9920        tran->tran_interconnect_type    = INTERCONNECT_FABRIC;

9922        tran->tran_pkt_constructor      = fcp_kmem_cache_constructor;
9923        tran->tran_pkt_destructor       = fcp_kmem_cache_destructor;
9924        tran->tran_setup_pkt            = fcp_pkt_setup;
9925        tran->tran_teardown_pkt         = fcp_pkt_teardown;
9926        tran->tran_hba_len              = pptr->port_priv_pkt_len +
9927            sizeof (struct fcp_pkt) + pptr->port_dmacookie_sz;
9928        if (pptr->port_state & FCP_STATE_FCA_IS_NODMA) {
9929                /*
9930                 * If FCA don't support DMA, then we use different vectors to
9931                 * minimize the effects on DMA code flow path
9932                 */
9933                tran->tran_start           = fcp_pseudo_start;
9934                tran->tran_init_pkt        = fcp_pseudo_init_pkt;
9935                tran->tran_destroy_pkt     = fcp_pseudo_destroy_pkt;
9936                tran->tran_sync_pkt        = fcp_pseudo_sync_pkt;
9937                tran->tran_dmafree         = fcp_pseudo_dmafree;
9938                tran->tran_setup_pkt       = NULL;
9939                tran->tran_teardown_pkt    = NULL;
9940                tran->tran_pkt_constructor = NULL;
9941                tran->tran_pkt_destructor  = NULL;
9942                pptr->port_data_dma_attr   = pseudo_fca_dma_attr;
9943        }

9945        /*
9946         * Allocate an ndi event handle
9947         */
9948        pptr->port_ndi_event_defs = (ndi_event_definition_t *)
9949            kmem_zalloc(sizeof (fcp_ndi_event_defs), KM_SLEEP);

9951        bcopy(fcp_ndi_event_defs, pptr->port_ndi_event_defs,
9952            sizeof (fcp_ndi_event_defs));

9954        (void) ndi_event_alloc_hdl(pptr->port_dip, NULL,
9955            &pptr->port_ndi_event_hdl, NDI_SLEEP);

9957        pptr->port_ndi_events.ndi_events_version = NDI_EVENTS_REV1;
9958        pptr->port_ndi_events.ndi_n_events = FCP_N_NDI_EVENTS;
9959        pptr->port_ndi_events.ndi_event_defs = pptr->port_ndi_event_defs;

9961        if (DEVI_IS_ATTACHING(pptr->port_dip) &&
9962            (ndi_event_bind_set(pptr->port_ndi_event_hdl,
```

```
9963             &pptr->port_ndi_events, NDI_SLEEP) != NDI_SUCCESS)) {
9964                 goto fail;
9965         }
9966         event_bind++;   /* Checked in fail case */

9968         if (scsi_hba_attach_setup(pptr->port_dip, &pptr->port_data_dma_attr,
9969             tran, SCSI_HBA_ADDR_COMPLEX | SCSI_HBA_TRAN_SCB)
9970             != DDI_SUCCESS) {
9971                 fcp_log(CE_WARN, pptr->port_dip,
9972                     "!fcp%d: scsi_hba_attach_setup failed", instance);
9973                 goto fail;
9974         }
9975         hba_attached++; /* Checked in fail case */

9977         pptr->port_mpxio = 0;
9978         if (mdi_phci_register(MDI_HCI_CLASS_SCSI, pptr->port_dip, 0) ==
9979             MDI_SUCCESS) {
9980                 pptr->port_mpxio++;
9981         }

9983         /*
9984          * The following code is putting the new port structure in the global
9985          * list of ports and, if it is the first port to attach, it start the
9986          * fcp_watchdog_tick.
9987          *
9988          * Why put this new port in the global before we are done attaching it?
9989          * We are actually making the structure globally known before we are
9990          * done attaching it.  The reason for that is: because of the code that
9991          * follows.  At this point the resources to handle the port are
9992          * allocated.  This function is now going to do the following:
9993          *
9994          *   1) It is going to try to register with the name server advertizing
9995          *      the new FCP capability of the port.
9996          *   2) It is going to play the role of the fp/fctl layer by building
9997          *      a list of worlwide names reachable through this port and call
9998          *      itself on fcp_statec_callback().  That requires the port to
9999          *      be part of the global list.
10000          */
10001         mutex_enter(&fcp_global_mutex);
10002         if (fcp_port_head == NULL) {
10003                 fcp_read_blacklist(pinfo->port_dip, &fcp_lun_blacklist);
10004         }
10005         pptr->port_next = fcp_port_head;
10006         fcp_port_head = pptr;
10007         soft_state_linked++;

10009         if (fcp_watchdog_init++ == 0) {
10010                 fcp_watchdog_tick = drv_sectohz(fcp_watchdog_timeout);
10010                 fcp_watchdog_tick = fcp_watchdog_timeout *
10011                     drv_usectohz(1000000);
10011                 fcp_watchdog_id = timeout(fcp_watch, NULL,
10012                     fcp_watchdog_tick);
10013         }
10014         mutex_exit(&fcp_global_mutex);

10016         /*
10017          * Here an attempt is made to register with the name server, the new
10018          * FCP capability.  That is done using an RTF_ID to the name server.
10019          * It is done synchronously.  The function fcp_do_ns_registry()
10020          * doesn't return till the name server responded.
10021          * On failures, just ignore it for now and it will get retried during
10022          * state change callbacks. We'll set a flag to show this failure
10023          */
10024         if (fcp_do_ns_registry(pptr, s_id)) {
10025                 mutex_enter(&pptr->port_mutex);
10026                 pptr->port_state |= FCP_STATE_NS_REG_FAILED;
```



```
10027                 mutex_exit(&pptr->port_mutex);
10028         } else {
10029                 mutex_enter(&pptr->port_mutex);
10030                 pptr->port_state &= ~(FCP_STATE_NS_REG_FAILED);
10031                 mutex_exit(&pptr->port_mutex);
10032         }

10034         /*
10035          * Lookup for boot WWN property
10036          */
10037         if (modrootloaded != 1) {
10038                 if ((ddi_prop_lookup_byte_array(DDI_DEV_T_ANY,
10039                     ddi_get_parent(pinfo->port_dip),
10040                     DDI_PROP_DONTPASS, OBP_BOOT_WWN,
10041                     &boot_wwn, &nbytes) == DDI_PROP_SUCCESS) &&
10042                     (nbytes == FC_WWN_SIZE)) {
10043                         bcopy(boot_wwn, pptr->port_boot_wwn, FC_WWN_SIZE);
10044                 }
10045                 if (boot_wwn) {
10046                         ddi_prop_free(boot_wwn);
10047                 }
10048         }

10050         /*
10051          * Handle various topologies and link states.
10052          */
10053         switch (FC_PORT_STATE_MASK(pptr->port_phys_state)) {
10054         case FC_STATE_OFFLINE:

10056                 /*
10057                  * we're attaching a port where the link is offline
10058                  *
10059                  * Wait for ONLINE, at which time a state
10060                  * change will cause a statec_callback
10061                  *
10062                  * in the mean time, do not do anything
10063                  */
10064                 res = DDI_SUCCESS;
10065                 pptr->port_state |= FCP_STATE_OFFLINE;
10066                 break;

10068         case FC_STATE_ONLINE: {
10069                 if (pptr->port_topology == FC_TOP_UNKNOWN) {
10070                         (void) fcp_linkreset(pptr, NULL, KM_NOSLEEP);
10071                         res = DDI_SUCCESS;
10072                         break;
10073                 }
10074                 /*
10075                  * discover devices and create nodes (a private
10076                  * loop or point-to-point)
10077                  */
10078                 ASSERT(pptr->port_topology != FC_TOP_UNKNOWN);

10080                 /*
10081                  * At this point we are going to build a list of all the ports
10082                  * that can be reached through this local port.  It looks like
10083                  * we cannot handle more than FCP_MAX_DEVICES per local port
10084                  * (128).
10085                  */
10086                 if ((tmp_list = (fc_portmap_t *)kmem_zalloc(
10087                     sizeof (fc_portmap_t) * FCP_MAX_DEVICES,
10088                     KM_NOSLEEP)) == NULL) {
10089                         fcp_log(CE_WARN, pptr->port_dip,
10090                             "!fcp%d: failed to allocate portmap",
10091                             instance);
10092                         goto fail;
```

```
10093                 }

10095                 /*
10096                  * fc_ulp_getportmap() is going to provide us with the list of
10097                  * remote ports in the buffer we just allocated.  The way the
10098                  * list is going to be retrieved depends on the topology.
10099                  * However, if we are connected to a Fabric, a name server
10100                  * request may be sent to get the list of FCP capable ports.
10101                  * It should be noted that is the case the request is
10102                  * synchronous.  This means we are stuck here till the name
10103                  * server replies.  A lot of things can change during that time
10104                  * and including, may be, being called on
10105                  * fcp_statec_callback() for different reasons. I'm not sure
10106                  * the code can handle that.
10107                  */
10108                 max_cnt = FCP_MAX_DEVICES;
10109                 alloc_cnt = FCP_MAX_DEVICES;
10110                 if ((res = fc_ulp_getportmap(pptr->port_fp_handle,
10111                     &tmp_list, &max_cnt, FC_ULP_PLOGI_PRESERVE)) !=
10112                     FC_SUCCESS) {
10113                         caddr_t msg;

10115                         (void) fc_ulp_error(res, &msg);

10117                         /*
10118                          * this  just means the transport is
10119                          * busy perhaps building a portmap so,
10120                          * for now, succeed this port attach
10121                          * when the transport has a new map,
10122                          * it'll send us a state change then
10123                          */
10124                         fcp_log(CE_WARN, pptr->port_dip,
10125                             "!failed to get port map : %s", msg);

10127                         res = DDI_SUCCESS;
10128                         break;  /* go return result */
10129                 }
10130                 if (max_cnt > alloc_cnt) {
10131                         alloc_cnt = max_cnt;
10132                 }

10134                 /*
10135                  * We are now going to call fcp_statec_callback() ourselves.
10136                  * By issuing this call we are trying to kick off the enumera-
10137                  * tion process.
10138                  */
10139                 /*
10140                  * let the state change callback do the SCSI device
10141                  * discovery and create the devinfos
10142                  */
10143                 fcp_statec_callback(ulph, pptr->port_fp_handle,
10144                     pptr->port_phys_state, pptr->port_topology, tmp_list,
10145                     max_cnt, pptr->port_id);

10147                 res = DDI_SUCCESS;
10148                 break;
10149         }

10151         default:
10152                 /* unknown port state */
10153                 fcp_log(CE_WARN, pptr->port_dip,
10154                     "!fcp%d: invalid port state at attach=0x%x",
10155                     instance, pptr->port_phys_state);

10157                 mutex_enter(&pptr->port_mutex);
10158                 pptr->port_phys_state = FCP_STATE_OFFLINE;
```



```
10159                 mutex_exit(&pptr->port_mutex);

10161                 res = DDI_SUCCESS;
10162                 break;
10163         }

10165         /* free temp list if used */
10166         if (tmp_list != NULL) {
10167                 kmem_free(tmp_list, sizeof (fc_portmap_t) * alloc_cnt);
10168         }

10170         /* note the attach time */
10171         pptr->port_attach_time = ddi_get_lbolt64();

10173         /* all done */
10174         return (res);

10176         /* a failure we have to clean up after */
10177 fail:
10178         fcp_log(CE_WARN, pptr->port_dip, "!failed to attach to port");

10180         if (soft_state_linked) {
10181                 /* remove this fcp_port from the linked list */
10182                 (void) fcp_soft_state_unlink(pptr);
10183         }

10185         /* unbind and free event set */
10186         if (pptr->port_ndi_event_hdl) {
10187                 if (event_bind) {
10188                         (void) ndi_event_unbind_set(pptr->port_ndi_event_hdl,
10189                             &pptr->port_ndi_events, NDI_SLEEP);
10190                 }
10191                 (void) ndi_event_free_hdl(pptr->port_ndi_event_hdl);
10192         }

10194         if (pptr->port_ndi_event_defs) {
10195                 (void) kmem_free(pptr->port_ndi_event_defs,
10196                     sizeof (fcp_ndi_event_defs));
10197         }

10199         /*
10200          * Clean up mpxio stuff
10201          */
10202         if (pptr->port_mpxio) {
10203                 (void) mdi_phci_unregister(pptr->port_dip, 0);
10204                 pptr->port_mpxio--;
10205         }

10207         /* undo SCSI HBA setup */
10208         if (hba_attached) {
10209                 (void) scsi_hba_detach(pptr->port_dip);
10210         }
10211         if (pptr->port_tran != NULL) {
10212                 scsi_hba_tran_free(pptr->port_tran);
10213         }

10215         mutex_enter(&fcp_global_mutex);

10217         /*
10218          * We check soft_state_linked, because it is incremented right before
10219          * we call increment fcp_watchdog_init.  Therefore, we know if
10220          * soft_state_linked is still FALSE, we do not want to decrement
10221          * fcp_watchdog_init or possibly call untimeout.
10222          */

10224         if (soft_state_linked) {
```

```
10225                 if (--fcp_watchdog_init == 0) {
10226                         timeout_id_t    tid = fcp_watchdog_id;

10228                         mutex_exit(&fcp_global_mutex);
10229                         (void) untimeout(tid);
10230                 } else {
10231                         mutex_exit(&fcp_global_mutex);
10232                 }
10233         } else {
10234                 mutex_exit(&fcp_global_mutex);
10235         }

10237         if (mutex_initted) {
10238                 mutex_destroy(&pptr->port_mutex);
10239                 mutex_destroy(&pptr->port_pkt_mutex);
10240         }

10242         if (tmp_list != NULL) {
10243                 kmem_free(tmp_list, sizeof (fc_portmap_t) * alloc_cnt);
10244         }

10246         /* this makes pptr invalid */
10247         ddi_soft_state_free(fcp_softstate, instance);

10249         return (DDI_FAILURE);
10250 }


10253 static int
10254 fcp_handle_port_detach(struct fcp_port *pptr, int flag, int instance)
10255 {
10256         int count = 0;

10258         mutex_enter(&pptr->port_mutex);

10260         /*
10261          * if the port is powered down or suspended, nothing else
10262          * to do; just return.
10263          */
10264         if (flag != FCP_STATE_DETACHING) {
10265                 if (pptr->port_state & (FCP_STATE_POWER_DOWN |
10266                     FCP_STATE_SUSPENDED)) {
10267                         pptr->port_state |= flag;
10268                         mutex_exit(&pptr->port_mutex);
10269                         return (FC_SUCCESS);
10270                 }
10271         }

10273         if (pptr->port_state & FCP_STATE_IN_MDI) {
10274                 mutex_exit(&pptr->port_mutex);
10275                 return (FC_FAILURE);
10276         }

10278         FCP_TRACE(fcp_logq, pptr->port_instbuf,
10279             fcp_trace, FCP_BUF_LEVEL_2, 0,
10280             "fcp_handle_port_detach: port is detaching");

10282         pptr->port_state |= flag;

10284         /*
10285          * Wait for any ongoing reconfig/ipkt to complete, that
10286          * ensures the freeing to targets/luns is safe.
10287          * No more ref to this port should happen from statec/ioctl
10288          * after that as it was removed from the global port list.
10289          */
10290         while (pptr->port_tmp_cnt || pptr->port_ipkt_cnt ||
```



```
10291             (pptr->port_state & FCP_STATE_IN_WATCHDOG)) {
10292                 /*
10293                  * Let's give sufficient time for reconfig/ipkt
10294                  * to complete.
10295                  */
10296                 if (count++ >= FCP_ICMD_DEADLINE) {
10297                         break;
10298                 }
10299                 mutex_exit(&pptr->port_mutex);
10300                 delay(drv_sectohz(1));
10301                 delay(drv_usectohz(1000000));
10301                 mutex_enter(&pptr->port_mutex);
10302         }

10304         /*
10305          * if the driver is still busy then fail to
10306          * suspend/power down.
10307          */
10308         if (pptr->port_tmp_cnt || pptr->port_ipkt_cnt ||
10309             (pptr->port_state & FCP_STATE_IN_WATCHDOG)) {
10310                 pptr->port_state &= ~flag;
10311                 mutex_exit(&pptr->port_mutex);
10312                 return (FC_FAILURE);
10313         }

10315         if (flag == FCP_STATE_DETACHING) {
10316                 pptr = fcp_soft_state_unlink(pptr);
10317                 ASSERT(pptr != NULL);
10318         }

10320         pptr->port_link_cnt++;
10321         pptr->port_state |= FCP_STATE_OFFLINE;
10322         pptr->port_state &= ~(FCP_STATE_ONLINING | FCP_STATE_ONLINE);

10324         fcp_update_state(pptr, (FCP_LUN_BUSY | FCP_LUN_MARK),
10325             FCP_CAUSE_LINK_DOWN);
10326         mutex_exit(&pptr->port_mutex);

10328         /* kill watch dog timer if we're the last */
10329         mutex_enter(&fcp_global_mutex);
10330         if (--fcp_watchdog_init == 0) {
10331                 timeout_id_t    tid = fcp_watchdog_id;
10332                 mutex_exit(&fcp_global_mutex);
10333                 (void) untimeout(tid);
10334         } else {
10335                 mutex_exit(&fcp_global_mutex);
10336         }

10338         /* clean up the port structures */
10339         if (flag == FCP_STATE_DETACHING) {
10340                 fcp_cleanup_port(pptr, instance);
10341         }

10343         return (FC_SUCCESS);
10344 }
______unchanged_portion_omitted_


12156 /*
12157  * called from fcp_port_attach() to resume a port
12158  * return DDI_* success/failure status
12159  * acquires and releases the global mutex
12160  * acquires and releases the port mutex
12161  */
12162 /*ARGSUSED*/
```

```
12164 static int
12165 fcp_handle_port_resume(opaque_t ulph, fc_ulp_port_info_t *pinfo,
12166     uint32_t s_id, fc_attach_cmd_t cmd, int instance)
12167 {
12168         int                     res = DDI_FAILURE; /* default result */
12169         struct fcp_port *pptr;          /* port state ptr */
12170         uint32_t                alloc_cnt;
12171         uint32_t                max_cnt;
12172         fc_portmap_t            *tmp_list = NULL;

12174         FCP_DTRACE(fcp_logq, "fcp", fcp_trace,
12175             FCP_BUF_LEVEL_8, 0, "port resume: for port %d",
12176             instance);

12178         if ((pptr = ddi_get_soft_state(fcp_softstate, instance)) == NULL) {
12179                 cmn_err(CE_WARN, "fcp: bad soft state");
12180                 return (res);
12181         }

12183         mutex_enter(&pptr->port_mutex);
12184         switch (cmd) {
12185         case FC_CMD_RESUME:
12186                 ASSERT((pptr->port_state & FCP_STATE_POWER_DOWN) == 0);
12187                 pptr->port_state &= ~FCP_STATE_SUSPENDED;
12188                 break;

12190         case FC_CMD_POWER_UP:
12191                 /*
12192                  * If the port is DDI_SUSPENded, defer rediscovery
12193                  * until DDI_RESUME occurs
12194                  */
12195                 if (pptr->port_state & FCP_STATE_SUSPENDED) {
12196                         pptr->port_state &= ~FCP_STATE_POWER_DOWN;
12197                         mutex_exit(&pptr->port_mutex);
12198                         return (DDI_SUCCESS);
12199                 }
12200                 pptr->port_state &= ~FCP_STATE_POWER_DOWN;
12201         }
12202         pptr->port_id = s_id;
12203         pptr->port_state = FCP_STATE_INIT;
12204         mutex_exit(&pptr->port_mutex);

12206         /*
12207          * Make a copy of ulp_port_info as fctl allocates
12208          * a temp struct.
12209          */
12210         (void) fcp_cp_pinfo(pptr, pinfo);

12212         mutex_enter(&fcp_global_mutex);
12213         if (fcp_watchdog_init++ == 0) {
12214                 fcp_watchdog_tick = drv_sectohz(fcp_watchdog_timeout);
12215                 fcp_watchdog_tick = fcp_watchdog_timeout *
12216                     drv_usectohz(1000000);
12215                 fcp_watchdog_id = timeout(fcp_watch,
12216                     NULL, fcp_watchdog_tick);
12217         }
12218         mutex_exit(&fcp_global_mutex);

12220         /*
12221          * Handle various topologies and link states.
12222          */
12223         switch (FC_PORT_STATE_MASK(pptr->port_phys_state)) {
12224         case FC_STATE_OFFLINE:
12225                 /*
12226                  * Wait for ONLINE, at which time a state
12227                  * change will cause a statec_callback
```



```
12228                  */
12229                 res = DDI_SUCCESS;
12230                 break;

12232         case FC_STATE_ONLINE:

12234                 if (pptr->port_topology == FC_TOP_UNKNOWN) {
12235                         (void) fcp_linkreset(pptr, NULL, KM_NOSLEEP);
12236                         res = DDI_SUCCESS;
12237                         break;
12238                 }

12240                 if (FC_TOP_EXTERNAL(pptr->port_topology) &&
12241                     !fcp_enable_auto_configuration) {
12242                         tmp_list = fcp_construct_map(pptr, &alloc_cnt);
12243                         if (tmp_list == NULL) {
12244                                 if (!alloc_cnt) {
12245                                         res = DDI_SUCCESS;
12246                                 }
12247                                 break;
12248                         }
12249                         max_cnt = alloc_cnt;
12250                 } else {
12251                         ASSERT(pptr->port_topology != FC_TOP_UNKNOWN);

12253                         alloc_cnt = FCP_MAX_DEVICES;

12255                         if ((tmp_list = (fc_portmap_t *)kmem_zalloc(
12256                             (sizeof (fc_portmap_t)) * alloc_cnt,
12257                             KM_NOSLEEP)) == NULL) {
12258                                 fcp_log(CE_WARN, pptr->port_dip,
12259                                     "!fcp%d: failed to allocate portmap",
12260                                     instance);
12261                                 break;
12262                         }

12264                         max_cnt = alloc_cnt;
12265                         if ((res = fc_ulp_getportmap(pptr->port_fp_handle,
12266                             &tmp_list, &max_cnt, FC_ULP_PLOGI_PRESERVE)) !=
12267                             FC_SUCCESS) {
12268                                 caddr_t msg;

12270                                 (void) fc_ulp_error(res, &msg);

12272                                 FCP_TRACE(fcp_logq, pptr->port_instbuf,
12273                                     fcp_trace, FCP_BUF_LEVEL_2, 0,
12274                                     "resume failed getportmap: reason=0x%x",
12275                                     res);

12277                                 fcp_log(CE_WARN, pptr->port_dip,
12278                                     "!failed to get port map : %s", msg);
12279                                 break;
12280                         }
12281                         if (max_cnt > alloc_cnt) {
12282                                 alloc_cnt = max_cnt;
12283                         }
12284                 }

12286                 /*
12287                  * do the SCSI device discovery and create
12288                  * the devinfos
12289                  */
12290                 fcp_statec_callback(ulph, pptr->port_fp_handle,
12291                     pptr->port_phys_state, pptr->port_topology, tmp_list,
12292                     max_cnt, pptr->port_id);
```

```
12294                 res = DDI_SUCCESS;
12295                 break;

12297         default:
12298                 fcp_log(CE_WARN, pptr->port_dip,
12299                     "!fcp%d: invalid port state at attach=0x%x",
12300                     instance, pptr->port_phys_state);

12302                 mutex_enter(&pptr->port_mutex);
12303                 pptr->port_phys_state = FCP_STATE_OFFLINE;
12304                 mutex_exit(&pptr->port_mutex);
12305                 res = DDI_SUCCESS;

12307                 break;
12308         }

12310         if (tmp_list != NULL) {
12311                 kmem_free(tmp_list, sizeof (fc_portmap_t) * alloc_cnt);
12312         }

12314         return (res);
12315 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   91358 Wed Aug 19 07:25:04 2015
new/usr/src/uts/common/io/fibre-channel/ulp/fcsm.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


741 /* ARGSUSED */
742 static int
743 fcsm_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
744 {
745         int     rval = DDI_SUCCESS;

747         switch (cmd) {
748         case DDI_DETACH: {
749                 fcsm_t  *fcsm;

751                 FCSM_DEBUG(SMDL_TRACE, (CE_CONT, SM_LOG, NULL, NULL,
752                     "detach: start. cmd <DETACH>", cmd));

754                 mutex_enter(&fcsm_global_mutex);

756                 /*
757                  * If port attach/detach in progress, then wait for 5 seconds
758                  * for them to complete.
759                  */
760                 if (fcsm_num_attaching || fcsm_num_detaching) {
761                         int count;

763                         FCSM_DEBUG(SMDL_TRACE, (CE_WARN, SM_LOG, NULL, NULL,
764                             "detach: wait for port attach/detach to complete"));

766                         count = 0;
767                         while ((count++ <= 30) &&
768                             (fcsm_num_attaching || fcsm_num_detaching)) {
769                                 mutex_exit(&fcsm_global_mutex);
770                                 delay(drv_sectohz(1));
770                                 delay(drv_usectohz(1000000));
771                                 mutex_enter(&fcsm_global_mutex);
772                         }

774                         /* Port attach/detach still in prog, so fail detach */
775                         if (fcsm_num_attaching || fcsm_num_detaching) {
776                                 mutex_exit(&fcsm_global_mutex);
777                                 FCSM_DEBUG(SMDL_ERR, (CE_WARN, SM_LOG, NULL,
778                                     NULL, "detach: Failing detach. port "
779                                     "attach/detach in progress"));
780                                 rval = DDI_FAILURE;
781                                 break;
782                         }
783                 }

785                 if (fcsm_port_head == NULL) {
786                         /* Not much do, Succeed to detach. */
787                         ddi_remove_minor_node(fcsm_dip, NULL);
788                         fcsm_dip = NULL;
789                         fcsm_detached = 0;
790                         mutex_exit(&fcsm_global_mutex);
791                         break;
792                 }

794                 /*
795                  * Check to see, if any ports are active.
796                  * If not, then set the DETACHING flag to indicate
797                  * that they are being detached.
```



```
798                  */
799                 fcsm = fcsm_port_head;
800                 while (fcsm != NULL) {

802                         mutex_enter(&fcsm->sm_mutex);
803                         if (!(fcsm->sm_flags & FCSM_ATTACHED) ||
804                             fcsm->sm_ncmds || fcsm->sm_cb_count) {
805                                 /* port is busy. We can't detach */
806                                 mutex_exit(&fcsm->sm_mutex);
807                                 break;
808                         }

810                         fcsm->sm_flags |= FCSM_DETACHING;
811                         mutex_exit(&fcsm->sm_mutex);

813                         fcsm = fcsm->sm_next;
814                 }

816                 /*
817                  * If all ports could not be marked for detaching,
818                  * then clear the flags and fail the detach.
819                  * Also if a port attach is currently in progress
820                  * then fail the detach.
821                  */
822                 if (fcsm != NULL || fcsm_num_attaching || fcsm_num_detaching) {
823                         /*
824                          * Some ports were busy, so can't detach.
825                          * Clear the DETACHING flag and return failure
826                          */
827                         fcsm = fcsm_port_head;
828                         while (fcsm != NULL) {
829                                 mutex_enter(&fcsm->sm_mutex);
830                                 if (fcsm->sm_flags & FCSM_DETACHING) {
831                                         fcsm->sm_flags &= ~FCSM_DETACHING;
832                                 }
833                                 mutex_exit(&fcsm->sm_mutex);

835                                 fcsm = fcsm->sm_next;
836                         }
837                         mutex_exit(&fcsm_global_mutex);
838                         return (DDI_FAILURE);
839                 } else {
840                         fcsm_detached = 1;
841                         /*
842                          * Mark all the detaching ports as detached, as we
843                          * will be detaching them
844                          */
845                         fcsm = fcsm_port_head;
846                         while (fcsm != NULL) {
847                                 mutex_enter(&fcsm->sm_mutex);
848                                 fcsm->sm_flags &= ~FCSM_DETACHING;
849                                 fcsm->sm_flags |= FCSM_DETACHED;
850                                 mutex_exit(&fcsm->sm_mutex);

852                                 fcsm = fcsm->sm_next;
853                         }
854                 }
855                 mutex_exit(&fcsm_global_mutex);


858                 /*
859                  * Go ahead and detach the ports
860                  */
861                 mutex_enter(&fcsm_global_mutex);
862                 while (fcsm_port_head != NULL) {
863                         fcsm = fcsm_port_head;
```

```
 864                         mutex_exit(&fcsm_global_mutex);

 866                         /*
 867                          * Call fcsm_cleanup_port(). This cleansup and
 868                          * removes the fcsm structure from global linked list
 869                          */
 870                         fcsm_cleanup_port(fcsm);

 872                         /*
 873                          * Soft state cleanup done.
 874                          * Remember that fcsm struct doesn't exist anymore.
 875                          */

 877                         mutex_enter(&fcsm_global_mutex);
 878                 }

 880                 ddi_remove_minor_node(fcsm_dip, NULL);
 881                 fcsm_dip = NULL;
 882                 mutex_exit(&fcsm_global_mutex);
 883                 break;
 884         }

 886         case DDI_SUSPEND:
 887                 rval = DDI_SUCCESS;
 888                 break;

 890         default:
 891                 FCSM_DEBUG(SMDL_ERR, (CE_NOTE, SM_LOG, NULL, NULL,
 892                     "detach: unknown cmd 0x%x", cmd));
 893                 rval = DDI_FAILURE;
 894                 break;
 895         }

 897         FCSM_DEBUG(SMDL_TRACE, (CE_CONT, SM_LOG, NULL, NULL,
 898             "detach: end. cmd 0x%x, rval 0x%x", cmd, rval));

 900         return (rval);
 901 }
```

*unchanged_portion_omitted_*

```
1007 static int
1008 fcsm_handle_port_detach(fc_ulp_port_info_t *pinfo, fcsm_t *fcsm,
1009     fc_detach_cmd_t cmd)
1010 {
1011         uint32_t        flag;
1012         int             count;
1013 #ifdef DEBUG
1014         char            pathname[MAXPATHLEN];
1015 #endif /* DEBUG */

1017         /*
1018          * If port is already powered down OR suspended and there is nothing
1019          * else to do then just return.
1020          * Otherwise, set the flag, so that no more new activity will be
1021          * initiated on this port.
1022          */
1023         mutex_enter(&fcsm->sm_mutex);

1025         switch (cmd) {
1026         case FC_CMD_DETACH:
1027                 flag = FCSM_DETACHING;
1028                 break;

1030         case FC_CMD_SUSPEND:
1031         case FC_CMD_POWER_DOWN:
```



```
1032                 ((cmd == FC_CMD_SUSPEND) ? (flag = FCSM_SUSPENDED) :
1033                     (flag = FCSM_POWER_DOWN));
1034                 if (fcsm->sm_flags &
1035                     (FCSM_POWER_DOWN | FCSM_SUSPENDED)) {
1036                         fcsm->sm_flags |= flag;
1037                         mutex_exit(&fcsm->sm_mutex);
1038                         return (DDI_SUCCESS);
1039                 }
1040                 break;

1042         default:
1043                 mutex_exit(&fcsm->sm_mutex);
1044                 return (DDI_FAILURE);
1045         };

1047         fcsm->sm_flags |= flag;

1049         /*
1050          * If some commands are pending OR callback in progress, then
1051          * wait for some finite amount of time for their completion.
1052          * TODO: add more checks here to check for cmd timeout, offline
1053          * timeout and other (??) threads.
1054          */
1055         count = 0;
1056         while ((count++ <= 30) && (fcsm->sm_ncmds || fcsm->sm_cb_count)) {
1057                 mutex_exit(&fcsm->sm_mutex);
1058                 delay(drv_sectohz(1));
1058                 delay(drv_usectohz(1000000));
1059                 mutex_enter(&fcsm->sm_mutex);
1060         }
1061         if (fcsm->sm_ncmds || fcsm->sm_cb_count) {
1062                 fcsm->sm_flags &= ~flag;
1063                 mutex_exit(&fcsm->sm_mutex);
1064                 fcsm_display(CE_WARN, SM_LOG, fcsm, NULL,
1065                     "port_detach: Failing suspend, port is busy");
1066                 return (DDI_FAILURE);
1067         }
1068         if (flag == FCSM_DETACHING) {
1069                 fcsm->sm_flags &= ~FCSM_DETACHING;
1070                 fcsm->sm_flags |= FCSM_DETACHED;
1071         }

1073         mutex_exit(&fcsm->sm_mutex);

1075         FCSM_DEBUG(SMDL_INFO, (CE_CONT, SM_LOG, fcsm, NULL,
1076             "port_detach: cmd 0x%x pathname <%s>",
1077             cmd, ddi_pathname(pinfo->port_dip, pathname)));

1079         if (cmd == FC_CMD_DETACH) {
1080                 fcsm_cleanup_port(fcsm);
1081                 /*
1082                  * Soft state cleanup done.
1083                  * Always remember that fcsm struct doesn't exist anymore.
1084                  */
1085         } else {
1086                 fcsm_suspend_port(fcsm);
1087         }

1089         return (DDI_SUCCESS);
1090 }
```

*unchanged_portion_omitted_*

```
1499 /* ARGSUSED */
1500 static int
1501 fcsm_fciocmd(intptr_t arg, int mode, cred_t *credp, fcio_t *fcio)
```

```
1502 {
1503         int  retval = 0;

1505         switch (fcio->fcio_cmd) {
1506         case  FCSMIO_CT_CMD: {
1507                 fcsm_t          *fcsm;
1508                 caddr_t         user_ibuf, user_obuf;
1509                 caddr_t         req_iu, rsp_iu, abuf;
1510                 int             status, instance, count;

1512                 if ((fcio->fcio_xfer != FCIO_XFER_RW) ||
1513                     (fcio->fcio_ilen == 0) || (fcio->fcio_ibuf == 0) ||
1514                     (fcio->fcio_olen == 0) || (fcio->fcio_obuf == 0) ||
1515                     (fcio->fcio_alen == 0) || (fcio->fcio_abuf == 0) ||
1516                     (fcio->fcio_flags != 0) || (fcio->fcio_cmd_flags != 0) ||
1517                     (fcio->fcio_ilen > FCSM_MAX_CT_SIZE) ||
1518                     (fcio->fcio_olen > FCSM_MAX_CT_SIZE) ||
1519                     (fcio->fcio_alen > MAXPATHLEN)) {
1520                         retval = EINVAL;
1521                         break;
1522                 }

1524                 /*
1525                  * Get the destination port for which this ioctl
1526                  * is targeted. The abuf will have the fp_minor
1527                  * number.
1528                  */
1529                 abuf = kmem_zalloc(fcio->fcio_alen, KM_SLEEP);
1530                 ASSERT(abuf != NULL);
1531                 if (ddi_copyin(fcio->fcio_abuf, abuf, fcio->fcio_alen, mode)) {
1532                         retval = EFAULT;
1533                         kmem_free(abuf, fcio->fcio_alen);
1534                         break;
1535                 }

1537                 instance = *((int *)abuf);
1538                 kmem_free(abuf, fcio->fcio_alen);

1540                 if (instance < 0) {
1541                         FCSM_DEBUG(SMDL_TRACE, (CE_WARN, SM_LOG, NULL, NULL,
1542                             "fciocmd: instance 0x%x, invalid instance",
1543                             instance));
1544                         retval = ENXIO;
1545                         break;
1546                 }

1548                 /*
1549                  * We confirmed that path corresponds to our port driver
1550                  * and a valid instance.
1551                  * If this port instance is not yet attached, then wait
1552                  * for a finite time for attach to complete
1553                  */
1554                 fcsm = ddi_get_soft_state(fcsm_state, instance);
1555                 count = 0;
1556                 while (count++ <= 30) {
1557                         if (fcsm != NULL) {
1558                                 mutex_enter(&fcsm->sm_mutex);
1559                                 if (fcsm->sm_flags & FCSM_ATTACHED) {
1560                                         mutex_exit(&fcsm->sm_mutex);
1561                                         break;
1562                                 }
1563                                 mutex_exit(&fcsm->sm_mutex);
1564                         }
1565                         if (count == 1) {
1566                                 FCSM_DEBUG(SMDL_TRACE,
1567                                     (CE_WARN, SM_LOG, NULL, NULL,
```



```
1568                                     "fciocmd: instance 0x%x, "
1569                                     "wait for port attach", instance));
1570                         }
1571                         delay(drv_sectohz(1));
1571                         delay(drv_usectohz(1000000));
1572                         fcsm = ddi_get_soft_state(fcsm_state, instance);
1573                 }
1574                 if (count > 30) {
1575                         FCSM_DEBUG(SMDL_TRACE, (CE_WARN, SM_LOG, NULL, NULL,
1576                             "fciocmd: instance 0x%x, port not attached",
1577                             instance));
1578                         retval = ENXIO;
1579                         break;
1580                 }

1582                 req_iu = kmem_zalloc(fcio->fcio_ilen, KM_SLEEP);
1583                 rsp_iu = kmem_zalloc(fcio->fcio_olen, KM_SLEEP);
1584                 ASSERT((req_iu != NULL) && (rsp_iu != NULL));

1586                 if (ddi_copyin(fcio->fcio_ibuf, req_iu,
1587                     fcio->fcio_ilen, mode)) {
1588                         retval = EFAULT;
1589                         kmem_free(req_iu, fcio->fcio_ilen);
1590                         kmem_free(rsp_iu, fcio->fcio_olen);
1591                         break;
1592                 }

1594                 user_ibuf = fcio->fcio_ibuf;
1595                 user_obuf = fcio->fcio_obuf;
1596                 fcio->fcio_ibuf = req_iu;
1597                 fcio->fcio_obuf = rsp_iu;

1599                 status = fcsm_ct_passthru(fcsm->sm_instance, fcio, KM_SLEEP,
1600                     FCSM_JOBFLAG_SYNC, NULL);
1601                 if (status != FC_SUCCESS) {
1602                         retval = EIO;
1603                 }

1605                 FCSM_DEBUG(SMDL_TRACE, (CE_CONT, SM_LOG, fcsm, NULL,
1606                     "fciocmd: cmd 0x%x completion status 0x%x",
1607                     fcio->fcio_cmd, status));
1608                 fcio->fcio_errno = status;
1609                 fcio->fcio_ibuf = user_ibuf;
1610                 fcio->fcio_obuf = user_obuf;

1612                 if (ddi_copyout(rsp_iu, fcio->fcio_obuf,
1613                     fcio->fcio_olen, mode)) {
1614                         retval = EFAULT;
1615                         kmem_free(req_iu, fcio->fcio_ilen);
1616                         kmem_free(rsp_iu, fcio->fcio_olen);
1617                         break;
1618                 }

1620                 kmem_free(req_iu, fcio->fcio_ilen);
1621                 kmem_free(rsp_iu, fcio->fcio_olen);

1623                 if (fcsm_fcio_copyout(fcio, arg, mode)) {
1624                         retval = EFAULT;
1625                 }
1626                 break;
1627         }

1629         case  FCSMIO_ADAPTER_LIST: {
1630                 fc_hba_list_t   *list;
1631                 int                     count;
```

```
1633                 if ((fcio->fcio_xfer != FCIO_XFER_RW) ||
1634                     (fcio->fcio_olen == 0) || (fcio->fcio_obuf == 0)) {
1635                         retval = EINVAL;
1636                         break;
1637                 }

1639                 list = kmem_zalloc(fcio->fcio_olen, KM_SLEEP);

1641                 if (ddi_copyin(fcio->fcio_obuf, list, fcio->fcio_olen, mode)) {
1642                         retval = EFAULT;
1643                         break;
1644                 }
1645                 list->version = FC_HBA_LIST_VERSION;

1647                 if (fcio->fcio_olen < MAXPATHLEN * list->numAdapters) {
1648                         retval = EFAULT;
1649                         break;
1650                 }

1652                 count = fc_ulp_get_adapter_paths((char *)list->hbaPaths,
1653                     list->numAdapters);
1654                 if (count < 0) {
1655                         /* Did something go wrong? */
1656                         FCSM_DEBUG(SMDL_TRACE, (CE_CONT, SM_LOG, NULL, NULL,
1657                             "Error fetching adapter list."));
1658                         retval = ENXIO;
1659                         kmem_free(list, fcio->fcio_olen);
1660                         break;
1661                 }
1662                 /* Sucess (or short buffer) */
1663                 list->numAdapters = count;
1664                 if (ddi_copyout(list, fcio->fcio_obuf,
1665                     fcio->fcio_olen, mode)) {
1666                         retval = EFAULT;
1667                 }
1668                 kmem_free(list, fcio->fcio_olen);
1669                 break;
1670         }

1672         default:
1673                 FCSM_DEBUG(SMDL_TRACE, (CE_NOTE, SM_LOG, NULL, NULL,
1674                     "fciocmd: unknown cmd <0x%x>", fcio->fcio_cmd));
1675                 retval = ENOTTY;
1676                 break;
1677         }

1679         return (retval);
1680 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   71160 Wed Aug 19 07:25:04 2015
new/usr/src/uts/common/io/gldutil.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1278 /* ========== */
1279 /* Token Ring */
1280 /* ========== */

1282 #define GLD_SR_VAR(macinfo)     \
1283         (((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->data)

1285 #define GLD_SR_HASH(macinfo)    ((struct srtab **)GLD_SR_VAR(macinfo))

1287 #define GLD_SR_MUTEX(macinfo)   \
1288         (&((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->datalock)

1290 static void gld_sr_clear(gld_mac_info_t *);
1291 static void gld_rcc_receive(gld_mac_info_t *, pktinfo_t *, struct gld_ri *,
1292     uchar_t *, int);
1293 static void gld_rcc_send(gld_mac_info_t *, queue_t *, uchar_t *,
1294     struct gld_ri **, uchar_t *);

1296 static mac_addr_t tokenbroadcastaddr2 = { 0xc0, 0x00, 0xff, 0xff, 0xff, 0xff };
1297 static struct gld_ri ri_ste_def;

1299 void
1300 gld_init_tr(gld_mac_info_t *macinfo)
1301 {
1302         struct gldkstats *sp =
1303             ((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->kstatp->ks_data;

1305         /* avoid endian-dependent code by initializing here instead of static */
1306         ri_ste_def.len = 2;
1307         ri_ste_def.rt = RT_STE;
1308         ri_ste_def.mtu = RT_MTU_MAX;
1309         ri_ste_def.dir = 0;
1310         ri_ste_def.res = 0;

1312         /* Assumptions we make for this medium */
1313         ASSERT(macinfo->gldm_type == DL_TPR);
1314         ASSERT(macinfo->gldm_addrlen == 6);
1315         ASSERT(macinfo->gldm_saplen == -2);
1316 #ifndef lint
1317         ASSERT(sizeof (struct tr_mac_frm_nori) == 14);
1318         ASSERT(sizeof (mac_addr_t) == 6);
1319 #endif

1321         mutex_init(GLD_SR_MUTEX(macinfo), NULL, MUTEX_DRIVER, NULL);

1323         GLD_SR_VAR(macinfo) = kmem_zalloc(sizeof (struct srtab *)*SR_HASH_SIZE,
1324                                 KM_SLEEP);

1326         /* Default is RDE enabled for this medium */
1327         ((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->rde_enabled =
1328             ddi_getprop(DDI_DEV_T_NONE, macinfo->gldm_devinfo, 0,
1329             "gld_rde_enable", 1);

1331         /*
1332          * Default is to use STE for unknown paths if RDE is enabled.
1333          * If RDE is disabled, default is to use NULL RIF fields.
1334          *
1335          * It's possible to force use of STE for ALL packets:
1336          * disable RDE but enable STE.  This may be useful for
```



```
1337          * non-transparent bridges, when it is not desired to run
1338          * the RDE algorithms.
1339          */
1340         ((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->rde_str_indicator_ste =
1341             ddi_getprop(DDI_DEV_T_NONE, macinfo->gldm_devinfo, 0,
1342             "gld_rde_str_indicator_ste",
1343             ((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->rde_enabled);

1345         /* Default 10 second route timeout on lack of activity */
1346         {
1347         int t = ddi_getprop(DDI_DEV_T_NONE, macinfo->gldm_devinfo, 0,
1348             "gld_rde_timeout", 10);
1349         if (t < 1)
1350                 t = 1;          /* Let's be reasonable */
1351         if (t > 600)
1352                 t = 600;        /* Let's be reasonable */
1353         /* We're using ticks (lbolts) for our timeout -- convert from seconds */
1354         t = drv_sectohz(t);
1354         t = drv_usectohz(1000000 * t);
1355         ((gld_mac_pvt_t *)macinfo->gldm_mac_pvt)->rde_timeout = t;
1356         }

1358         kstat_named_init(&sp->glds_dot5_line_error,
1359             "line_errors", KSTAT_DATA_UINT32);
1360         kstat_named_init(&sp->glds_dot5_burst_error,
1361             "burst_errors", KSTAT_DATA_UINT32);
1362         kstat_named_init(&sp->glds_dot5_signal_loss,
1363             "signal_losses", KSTAT_DATA_UINT32);

1365         /*
1366          * only initialize the new statistics if the driver
1367          * knows about them.
1368          */
1369         if (macinfo->gldm_driver_version != GLD_VERSION_200)
1370                 return;

1372         kstat_named_init(&sp->glds_dot5_ace_error,
1373             "ace_errors", KSTAT_DATA_UINT32);
1374         kstat_named_init(&sp->glds_dot5_internal_error,
1375             "internal_errors", KSTAT_DATA_UINT32);
1376         kstat_named_init(&sp->glds_dot5_lost_frame_error,
1377             "lost_frame_errors", KSTAT_DATA_UINT32);
1378         kstat_named_init(&sp->glds_dot5_frame_copied_error,
1379             "frame_copied_errors", KSTAT_DATA_UINT32);
1380         kstat_named_init(&sp->glds_dot5_token_error,
1381             "token_errors", KSTAT_DATA_UINT32);
1382         kstat_named_init(&sp->glds_dot5_freq_error,
1383             "freq_errors", KSTAT_DATA_UINT32);
1384 }
______unchanged_portion_omitted_
```

****
121177 Wed Aug 19 07:25:04 2015
new/usr/src/uts/common/io/hxge/hxge_main.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
2399 /*
2400  *      hxge_m_start() -- start transmitting and receiving.
2401  *
2402  *      This function is called by the MAC layer when the first
2403  *      stream is open to prepare the hardware ready for sending
2404  *      and transmitting packets.
2405  */
2406 static int
2407 hxge_m_start(void *arg)
2408 {
2409         p_hxge_t hxgep = (p_hxge_t)arg;

2411         HXGE_DEBUG_MSG((hxgep, NEMO_CTL, "==> hxge_m_start"));

2413         MUTEX_ENTER(hxgep->genlock);

2415         if (hxge_init(hxgep) != DDI_SUCCESS) {
2416                 HXGE_ERROR_MSG((hxgep, HXGE_ERR_CTL,
2417                     "<== hxge_m_start: initialization failed"));
2418                 MUTEX_EXIT(hxgep->genlock);
2419                 return (EIO);
2420         }

2422         if (hxgep->hxge_mac_state != HXGE_MAC_STARTED) {
2423                 /*
2424                  * Start timer to check the system error and tx hangs
2425                  */
2426                 hxgep->hxge_timerid = hxge_start_timer(hxgep,
2427                     hxge_check_hw_state, HXGE_CHECK_TIMER);

2429                 hxgep->hxge_mac_state = HXGE_MAC_STARTED;

2431                 hxgep->timeout.link_status = 0;
2432                 hxgep->timeout.report_link_status = B_TRUE;
2433                 hxgep->timeout.ticks = drv_sectohz(2);
2433                 hxgep->timeout.ticks = drv_usectohz(2 * 1000000);

2435                 /* Start the link status timer to check the link status */
2436                 MUTEX_ENTER(&hxgep->timeout.lock);
2437                 hxgep->timeout.id = timeout(hxge_link_poll, (void *)hxgep,
2438                     hxgep->timeout.ticks);
2439                 MUTEX_EXIT(&hxgep->timeout.lock);
2440         }

2442         MUTEX_EXIT(hxgep->genlock);

2444         HXGE_DEBUG_MSG((hxgep, NEMO_CTL, "<== hxge_m_start"));

2446         return (0);
2447 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28739 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_stats.c
XXXX introduce drv_sectohz
**********************************************************
```

_____*unchanged_portion_omitted_*

```
 734 /*
 735  * hermon_kstat_perfcntr64_update_thread()
 736  *    Context: Entry point for a kernel thread
 737  *
 738  * Maintain 64 bit performance counters in software using the 32 bit
 739  * hardware counters.
 740  */
 741 static void
 742 hermon_kstat_perfcntr64_update_thread(void *arg)
 743 {
 744         hermon_state_t          *state = (hermon_state_t *)arg;
 745         hermon_ks_info_t        *ksi = state->hs_ks_info;
 746         uint_t                  i;
 747         clock_t                 delta = drv_sectohz(1);
 747         clock_t                 delta = drv_usectohz(1000000);

 749         mutex_enter(&ksi->hki_perfcntr64_lock);
 750         /*
 751          * Every one second update the values 64 bit software counters
 752          * for all ports. Exit if HERMON_PERFCNTR64_THREAD_EXIT flag is set.
 753          */
 754         while (!(ksi->hki_perfcntr64_flags & HERMON_PERFCNTR64_THREAD_EXIT)) {
 755                 for (i = 0; i < state->hs_cfg_profile->cp_num_ports; i++) {
 756                         if (ksi->hki_perfcntr64[i].hki64_enabled) {
 757                                 (void) hermon_kstat_perfcntr64_read(state,
 758                                     i + 1, 1);
 759                         }
 760                 }
 761                 /* sleep for a second */
 762                 (void) cv_reltimedwait(&ksi->hki_perfcntr64_cv,
 763                     &ksi->hki_perfcntr64_lock, delta, TR_CLOCK_TICK);
 764         }
 765         ksi->hki_perfcntr64_flags = 0;
 766         mutex_exit(&ksi->hki_perfcntr64_lock);
 767 }
```

_____*unchanged_portion_omitted_*

****
25778 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_stats.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 655 /*
 656  * tavor_kstat_perfcntr64_update_thread()
 657  *    Context: Entry point for a kernel thread
 658  *
 659  * Maintain 64 bit performance counters in software using the 32 bit
 660  * hardware counters.
 661  */
 662 static void
 663 tavor_kstat_perfcntr64_update_thread(void *arg)
 664 {
 665         tavor_state_t           *state = (tavor_state_t *)arg;
 666         tavor_ks_info_t         *ksi = state->ts_ks_info;
 667         uint_t                  i;

 669         mutex_enter(&ksi->tki_perfcntr64_lock);
 670         /*
 671          * Every one second update the values 64 bit software counters
 672          * for all ports. Exit if TAVOR_PERFCNTR64_THREAD_EXIT flag is set.
 673          */
 674         while (!(ksi->tki_perfcntr64_flags & TAVOR_PERFCNTR64_THREAD_EXIT)) {
 675                 for (i = 0; i < state->ts_cfg_profile->cp_num_ports; i++) {
 676                         if (ksi->tki_perfcntr64[i].tki64_enabled) {
 677                                 (void) tavor_kstat_perfcntr64_read(state,
 678                                     i + 1, 1);
 679                         }
 680                 }
 681                 /* sleep for a second */
 682                 (void) cv_timedwait(&ksi->tki_perfcntr64_cv,
 683                     &ksi->tki_perfcntr64_lock,
 684                     ddi_get_lbolt() + drv_sectohz(1));
 684                     ddi_get_lbolt() + drv_usectohz(1000000));
 685         }
 686         ksi->tki_perfcntr64_flags = 0;
 687         mutex_exit(&ksi->tki_perfcntr64_lock);
 688 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31154 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/clients/rds/rdsib_buf.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 112 void
 113 rds_free_recv_caches(rds_state_t *statep)
 114 {
 115         rds_hca_t       *hcap;
 116         int             ret;

 118         RDS_DPRINTF4("rds_free_recv_caches", "Enter");

 120         mutex_enter(&rds_dpool.pool_lock);
 121         if (rds_dpool.pool_memp == NULL) {
 122                 RDS_DPRINTF2("rds_free_recv_caches", "Caches are empty");
 123                 mutex_exit(&rds_dpool.pool_lock);
 124                 return;
 125         }

 127         /*
 128          * All buffers must have been freed as all sessions are closed
 129          * and destroyed
 130          */
 131         ASSERT(rds_dpool.pool_nbusy == 0);
 132         RDS_DPRINTF2("rds_free_recv_caches", "Data Pool has "
 133             "pending buffers: %d", rds_dpool.pool_nbusy);
 134         while (rds_dpool.pool_nbusy != 0) {
 135                 mutex_exit(&rds_dpool.pool_lock);
 136                 delay(drv_sectohz(1));
 136                 delay(drv_usectohz(1000000));
 137                 mutex_enter(&rds_dpool.pool_lock);
 138         }

 140         hcap = statep->rds_hcalistp;
 141         while (hcap != NULL) {
 142                 if (hcap->hca_mrhdl != NULL) {
 143                         ret = ibt_deregister_mr(hcap->hca_hdl,
 144                             hcap->hca_mrhdl);
 145                         if (ret == IBT_SUCCESS) {
 146                                 hcap->hca_mrhdl = NULL;
 147                                 hcap->hca_lkey = 0;
 148                                 hcap->hca_rkey = 0;
 149                         } else {
 150                                 RDS_DPRINTF2(LABEL, "ibt_deregister_mr "
 151                                     "failed: %d, mrhdl: 0x%p", ret,
 152                                     hcap->hca_mrhdl);
 153                         }
 154                 }
 155                 hcap = hcap->hca_nextp;
 156         }

 158         kmem_free(rds_dpool.pool_bufmemp, (rds_dpool.pool_nbuffers +
 159             rds_cpool.pool_nbuffers) * sizeof (rds_buf_t));
 160         rds_dpool.pool_bufmemp = NULL;

 162         kmem_free(rds_dpool.pool_memp, rds_dpool.pool_memsize);
 163         rds_dpool.pool_memp = NULL;

 165         mutex_exit(&rds_dpool.pool_lock);

 167         RDS_DPRINTF4("rds_free_recv_caches", "Return");
 168 }
______unchanged_portion_omitted_
```



```
 802 boolean_t
 803 rds_is_recvq_empty(rds_ep_t *ep, boolean_t wait)
 804 {
 805         rds_qp_t        *recvqp;
 806         rds_bufpool_t   *rpool;
 807         boolean_t ret = B_TRUE;

 809         recvqp = &ep->ep_recvqp;
 810         mutex_enter(&recvqp->qp_lock);
 811         RDS_DPRINTF2("rds_is_recvq_empty", "EP(%p): QP has %d WRs",
 812             ep, recvqp->qp_level);
 813         if (wait) {
 814                 /* wait until the RQ is empty */
 815                 while (recvqp->qp_level != 0) {
 816                         /* wait one second and try again */
 817                         mutex_exit(&recvqp->qp_lock);
 818                         delay(drv_sectohz(1));
 818                         delay(drv_usectohz(1000000));
 819                         mutex_enter(&recvqp->qp_lock);
 820                 }
 821         } else if (recvqp->qp_level != 0) {
 822                         ret = B_FALSE;
 823         }
 824         mutex_exit(&recvqp->qp_lock);

 826         rpool = &ep->ep_rcvpool;
 827         mutex_enter(&rpool->pool_lock);

 829         /*
 830          * During failovers/reconnects, the app may still have some buffers
 831          * on thier socket queues. Waiting here for those buffers may
 832          * cause a hang. It seems ok for those buffers to get freed later.
 833          */
 834         if (rpool->pool_nbusy != 0) {
 835                 RDS_DPRINTF2("rds_is_recvq_empty", "EP(%p): "
 836                     "There are %d pending buffers on sockqs", ep,
 837                     rpool->pool_nbusy);
 838                 ret = B_FALSE;
 839         }
 840         mutex_exit(&rpool->pool_lock);

 842         return (ret);
 843 }

 845 boolean_t
 846 rds_is_sendq_empty(rds_ep_t *ep, uint_t wait)
 847 {
 848         rds_bufpool_t   *spool;
 849         rds_buf_t       *bp;
 850         boolean_t       ret1 = B_TRUE;

 852         /* check if all the sends completed */
 853         spool = &ep->ep_sndpool;
 854         mutex_enter(&spool->pool_lock);
 855         RDS_DPRINTF2("rds_is_sendq_empty", "EP(%p): "
 856             "Send Pool contains: %d", ep, spool->pool_nbusy);
 857         if (wait) {
 858                 while (spool->pool_nbusy != 0) {
 859                         if (rds_no_interrupts) {
 860                                 /* wait one second and try again */
 861                                 delay(drv_sectohz(1));
 861                                 delay(drv_usectohz(1000000));
 862                                 rds_poll_send_completions(ep->ep_sendcq, ep,
 863                                     B_TRUE);
 864                         } else {
```

```
865                                 /* wait one second and try again */
866                                 mutex_exit(&spool->pool_lock);
867                                 delay(drv_sectohz(1));
867                                 delay(drv_usectohz(1000000));
868                                 mutex_enter(&spool->pool_lock);
869                         }
870                 }

872                 if ((wait == 2) && (ep->ep_type == RDS_EP_TYPE_DATA)) {
873                         rds_buf_t       *ackbp;
874                         rds_buf_t       *prev_ackbp;

876                         /*
877                          * If the last one is acknowledged then everything
878                          * is acknowledged
879                          */
880                         bp = spool->pool_tailp;
881                         ackbp = *(rds_buf_t **)ep->ep_ack_addr;
882                         prev_ackbp = ackbp;
883                         RDS_DPRINTF2("rds_is_sendq_empty", "EP(%p): "
884                             "Checking for acknowledgements", ep);
885                         while (bp != ackbp) {
886                                 RDS_DPRINTF2("rds_is_sendq_empty",
887                                     "EP(%p) BP(0x%p/0x%p) last "
888                                     "sent/acknowledged", ep, bp, ackbp);
889                                 mutex_exit(&spool->pool_lock);
890                                 delay(drv_sectohz(1));
890                                 delay(drv_usectohz(1000000));
891                                 mutex_enter(&spool->pool_lock);

893                                 bp = spool->pool_tailp;
894                                 ackbp = *(rds_buf_t **)ep->ep_ack_addr;
895                                 if (ackbp == prev_ackbp) {
896                                         RDS_DPRINTF2("rds_is_sendq_empty",
897                                             "There has been no progress,"
898                                             "give up and proceed");
899                                         break;
900                                 }
901                                 prev_ackbp = ackbp;
902                         }
903                 }
904         } else if (spool->pool_nbusy != 0) {
905                         ret1 = B_FALSE;
906         }
907         mutex_exit(&spool->pool_lock);

909         /* check if all the rdma acks completed */
910         mutex_enter(&ep->ep_lock);
911         RDS_DPRINTF2("rds_is_sendq_empty", "EP(%p): "
912             "Outstanding RDMA Acks: %d", ep, ep->ep_rdmacnt);
913         if (wait) {
914                 while (ep->ep_rdmacnt != 0) {
915                         if (rds_no_interrupts) {
916                                 /* wait one second and try again */
917                                 delay(drv_sectohz(1));
917                                 delay(drv_usectohz(1000000));
918                                 rds_poll_send_completions(ep->ep_sendcq, ep,
919                                     B_FALSE);
920                         } else {
921                                 /* wait one second and try again */
922                                 mutex_exit(&ep->ep_lock);
923                                 delay(drv_sectohz(1));
923                                 delay(drv_usectohz(1000000));
924                                 mutex_enter(&ep->ep_lock);
925                         }
926                 }
```



```
927         } else if (ep->ep_rdmacnt != 0) {
928                         ret1 = B_FALSE;
929         }
930         mutex_exit(&ep->ep_lock);

932         return (ret1);
933 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28874 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/clients/rds/rdsib_cm.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2005 SilverStorm Technologies, Inc. All rights reserved.
 *
 * This software is available to you under a choice of one of two
 * licenses.  You may choose to be licensed under the terms of the GNU
 * General Public License (GPL) Version 2, available from the file
 * COPYING in the main directory of this source tree, or the
 * OpenIB.org BSD license below:
 *
 *     Redistribution and use in source and binary forms, with or
 *     without modification, are permitted provided that the following
 *     conditions are met:
 *
 *      - Redistributions of source code must retain the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer.
 *
 *      - Redistributions in binary form must reproduce the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer in the documentation and/or other materials
 *        provided with the distribution.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */
/*
 * Sun elects to include this software in Sun product
 * under the OpenIB BSD license.
 *
 *
```



```
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#include <sys/ib/clients/rds/rdsib_cm.h>
#include <sys/ib/clients/rds/rdsib_ib.h>
#include <sys/ib/clients/rds/rdsib_buf.h>
#include <sys/ib/clients/rds/rdsib_ep.h>

/*
 * This file contains CM related work:
 *
 * Service registration/deregistration
 * Path lookup
 * CM connection callbacks
 * CM active and passive connection establishment
 * Connection failover
 */

#define SRCIP   src_addr.un.ip4addr
#define DSTIP   dst_addr.un.ip4addr

/*
 * Handle an incoming CM REQ
 */
/* ARGSUSED */
static ibt_cm_status_t
rds_handle_cm_req(rds_state_t *statep, ibt_cm_event_t *evp,
    ibt_cm_return_args_t *rargsp, void *rcmp, ibt_priv_data_len_t rcmp_len)
{
        ibt_cm_req_rcv_t        *reqp;
        ib_gid_t                lgid, rgid;
        rds_cm_private_data_t   cmp;
        rds_session_t           *sp;
        rds_ep_t                *ep;
        ibt_channel_hdl_t       chanhdl;
        ibt_ip_cm_info_t        ipcm_info;
        uint8_t                 save_state, save_type;
        int                     ret;

        RDS_DPRINTF2("rds_handle_cm_req", "Enter");

        reqp = &evp->cm_event.req;
        rgid = reqp->req_prim_addr.av_dgid; /* requester gid */
        lgid = reqp->req_prim_addr.av_sgid; /* receiver gid */

        RDS_DPRINTF2(LABEL, "REQ Received: From: %llx:%llx To: %llx:%llx",
            rgid.gid_prefix, rgid.gid_guid, lgid.gid_prefix, lgid.gid_guid);

        /*
         * CM private data brings IP information
         * Private data received is a stream of bytes and may not be properly
         * aligned. So, bcopy the data onto the stack before accessing it.
         */
        bcopy((uint8_t *)evp->cm_priv_data, &cmp,
            sizeof (rds_cm_private_data_t));
```

```
128         /* extract the CM IP info */
129         ret = ibt_get_ip_data(evp->cm_priv_data_len, evp->cm_priv_data,
130             &ipcm_info);
131         if (ret != IBT_SUCCESS) {
132                 RDS_DPRINTF2("rds_handle_cm_req", "ibt_get_ip_data failed: %d",
133                     ret);
134                 return (IBT_CM_REJECT);
135         }

137         RDS_DPRINTF2("rds_handle_cm_req",
138             "REQ Received: From IP: 0x%x To IP: 0x%x type: %d",
139             ipcm_info.SRCIP, ipcm_info.DSTIP, cmp.cmp_eptype);

141         if (cmp.cmp_version != RDS_VERSION) {
142                 RDS_DPRINTF2(LABEL, "Version Mismatch: Local version: %d "
143                     "Remote version: %d", RDS_VERSION, cmp.cmp_version);
144                 return (IBT_CM_REJECT);
145         }

147         /* RDS supports V4 addresses only */
148         if ((ipcm_info.src_addr.family != AF_INET) ||
149             (ipcm_info.dst_addr.family != AF_INET)) {
150                 RDS_DPRINTF2(LABEL, "Unsupported Address Family: "
151                     "src: %d dst: %d", ipcm_info.src_addr.family,
152                     ipcm_info.dst_addr.family);
153                 return (IBT_CM_REJECT);
154         }

156         if (cmp.cmp_arch != RDS_THIS_ARCH) {
157                 RDS_DPRINTF2(LABEL, "ARCH does not match (%d != %d)",
158                     cmp.cmp_arch, RDS_THIS_ARCH);
159                 return (IBT_CM_REJECT);
160         }

162         if ((cmp.cmp_eptype != RDS_EP_TYPE_CTRL) &&
163             (cmp.cmp_eptype != RDS_EP_TYPE_DATA)) {
164                 RDS_DPRINTF2(LABEL, "Unknown Channel type: %d", cmp.cmp_eptype);
165                 return (IBT_CM_REJECT);
166         }

168         /* user_buffer_size should be same on all nodes */
169         if (cmp.cmp_user_buffer_size != UserBufferSize) {
170                 RDS_DPRINTF2(LABEL,
171                     "UserBufferSize Mismatch, this node: %d remote node: %d",
172                     UserBufferSize, cmp.cmp_user_buffer_size);
173                 return (IBT_CM_REJECT);
174         }

176         /*
177          * RDS needs more time to process a failover REQ so send an MRA.
178          * Otherwise, the remote may retry the REQ and fail the connection.
179          */
180         if ((cmp.cmp_failover) && (cmp.cmp_eptype == RDS_EP_TYPE_DATA)) {
181                 RDS_DPRINTF2("rds_handle_cm_req", "Session Failover, send MRA");
182                 (void) ibt_cm_delay(IBT_CM_DELAY_REQ, evp->cm_session_id,
183                     10000000 /* 10 sec */, NULL, 0);
184         }

186         /* Is there a session to the destination node? */
187         rw_enter(&statep->rds_sessionlock, RW_READER);
188         sp = rds_session_lkup(statep, ipcm_info.SRCIP, rgid.gid_guid);
189         rw_exit(&statep->rds_sessionlock);

191         if (sp == NULL) {
192                 /*
193                  * currently there is no session to the destination
```



```
194                  * remote ip in the private data is the local ip and vice
195                  * versa
196                  */
197                 sp = rds_session_create(statep, ipcm_info.DSTIP,
198                     ipcm_info.SRCIP, reqp, RDS_SESSION_PASSIVE);
199                 if (sp == NULL) {
200                         /* Check the list anyway. */
201                         rw_enter(&statep->rds_sessionlock, RW_READER);
202                         sp = rds_session_lkup(statep, ipcm_info.SRCIP,
203                             rgid.gid_guid);
204                         rw_exit(&statep->rds_sessionlock);
205                         if (sp == NULL) {
206                                 /*
207                                  * The only way this can fail is due to lack
208                                  * of kernel resources
209                                  */
210                                 return (IBT_CM_REJECT);
211                         }
212                 }
213         }

215         rw_enter(&sp->session_lock, RW_WRITER);

217         /* catch peer-to-peer case as soon as possible */
218         if ((sp->session_state == RDS_SESSION_STATE_CREATED) ||
219             (sp->session_state == RDS_SESSION_STATE_INIT)) {
220                 /* Check possible peer-to-peer case here */
221                 if (sp->session_type != RDS_SESSION_PASSIVE) {
222                         RDS_DPRINTF2("rds_handle_cm_req",
223                             "SP(%p) Peer-peer connection handling", sp);
224                         if (lgid.gid_guid > rgid.gid_guid) {
225                                 /* this node is active so reject this request */
226                                 rw_exit(&sp->session_lock);
227                                 return (IBT_CM_REJECT);
228                         } else {
229                                 /* this node is passive, change the session */
230                                 sp->session_type = RDS_SESSION_PASSIVE;
231                                 sp->session_lgid = lgid;
232                                 sp->session_rgid = rgid;
233                         }
234                 }
235         }

237         RDS_DPRINTF2(LABEL, "SP(%p) state: %d", sp, sp->session_state);
238         save_state = sp->session_state;
239         save_type = sp->session_type;

241         switch (sp->session_state) {
242         case RDS_SESSION_STATE_CONNECTED:
243                 RDS_DPRINTF2(LABEL, "STALE Session Detected SP(%p)", sp);
244                 sp->session_state = RDS_SESSION_STATE_ERROR;
245                 RDS_DPRINTF3("rds_handle_cm_req", "SP(%p) State "
246                     "RDS_SESSION_STATE_ERROR", sp);

248                 /* FALLTHRU */
249         case RDS_SESSION_STATE_ERROR:
250         case RDS_SESSION_STATE_PASSIVE_CLOSING:
251                 /*
252                  * Some other thread must be processing this session,
253                  * this thread must wait until the other thread finishes.
254                  */
255                 sp->session_type = RDS_SESSION_PASSIVE;
256                 rw_exit(&sp->session_lock);

258                 /* Handling this will take some time, so send an MRA */
259                 (void) ibt_cm_delay(IBT_CM_DELAY_REQ, evp->cm_session_id,
```

```
260                     10000000 /* 10 sec */, NULL, 0);

262             /*
263              * Any pending completions don't get flushed until the channel
264              * is closed. So, passing 0 here will not wait for pending
265              * completions in rds_session_close before closing the channel
266              */
267             rds_session_close(sp, IBT_NOCALLBACKS, 0);

269             rw_enter(&sp->session_lock, RW_WRITER);

271             /*
272              * If the session was in ERROR, then either a failover thread
273              * or event_failure thread would be processing this session.
274              * This thread should wait for event_failure thread to
275              * complete. This need not wait for failover thread.
276              */
277             if ((save_state != RDS_SESSION_STATE_CONNECTED) &&
278                 (save_type == RDS_SESSION_PASSIVE)) {
279                     /*
280                      * The other thread is event_failure thread,
281                      * wait until it finishes.
282                      */
283                     while (!((sp->session_state ==
284                         RDS_SESSION_STATE_FAILED) ||
285                         (sp->session_state ==
286                         RDS_SESSION_STATE_FINI))) {
287                             rw_exit(&sp->session_lock);
288                             delay(drv_sectohz(1));
288                             delay(drv_usectohz(1000000));
289                             rw_enter(&sp->session_lock, RW_WRITER);
290                     }
291             }

293             /* move the session to init state */
294             if ((sp->session_state == RDS_SESSION_STATE_ERROR) ||
295                 (sp->session_state == RDS_SESSION_STATE_PASSIVE_CLOSING)) {
296                     ret = rds_session_reinit(sp, lgid);
297                     sp->session_myip = ipcm_info.DSTIP;
298                     sp->session_lgid = lgid;
299                     sp->session_rgid = rgid;
300                     if (ret != 0) {
301                             rds_session_fini(sp);
302                             sp->session_state = RDS_SESSION_STATE_FAILED;
303                             RDS_DPRINTF3("rds_handle_cm_req",
304                                 "SP(%p) State RDS_SESSION_STATE_FAILED",
305                                 sp);
306                             rw_exit(&sp->session_lock);
307                             return (IBT_CM_REJECT);
308                     } else {
309                             sp->session_state = RDS_SESSION_STATE_INIT;
310                             RDS_DPRINTF3("rds_handle_cm_req",
311                                 "SP(%p) State RDS_SESSION_STATE_INIT", sp);
312                     }

314                     if (cmp.cmp_eptype == RDS_EP_TYPE_CTRL) {
315                             ep = &sp->session_ctrlep;
316                     } else {
317                             ep = &sp->session_dataep;
318                     }
319                     break;
320             }

322             /* FALLTHRU */
323         case RDS_SESSION_STATE_CREATED:
324         case RDS_SESSION_STATE_FAILED:
```



```
325         case RDS_SESSION_STATE_FINI:
326                 /*
327                  * Initialize both channels, we accept this connection
328                  * only if both channels are initialized
329                  */
330                 sp->session_type = RDS_SESSION_PASSIVE;
331                 sp->session_lgid = lgid;
332                 sp->session_rgid = rgid;
333                 sp->session_state = RDS_SESSION_STATE_CREATED;
334                 RDS_DPRINTF3("rds_handle_cm_req", "SP(%p) State "
335                     "RDS_SESSION_STATE_CREATED", sp);
336                 ret = rds_session_init(sp);
337                 if (ret != 0) {
338                         /* Seems like there are not enough resources */
339                         sp->session_state = RDS_SESSION_STATE_FAILED;
340                         RDS_DPRINTF3("rds_handle_cm_req", "SP(%p) State "
341                             "RDS_SESSION_STATE_FAILED", sp);
342                         rw_exit(&sp->session_lock);
343                         return (IBT_CM_REJECT);
344                 }
345                 sp->session_state = RDS_SESSION_STATE_INIT;
346                 RDS_DPRINTF3("rds_handle_cm_req", "SP(%p) State "
347                     "RDS_SESSION_STATE_INIT", sp);

349                 /* FALLTHRU */
350         case RDS_SESSION_STATE_INIT:
351                 /*
352                  * When re-using an existing session, make sure the
353                  * session is still through the same HCA. Otherwise, the
354                  * memory registrations have to moved to the new HCA.
355                  */
356                 if (cmp.cmp_eptype == RDS_EP_TYPE_DATA) {
357                         if (sp->session_lgid.gid_guid != lgid.gid_guid) {
358                                 RDS_DPRINTF2("rds_handle_cm_req",
359                                     "Existing Session but different gid "
360                                     "existing: 0x%llx, new: 0x%llx, "
361                                     "sending an MRA",
362                                     sp->session_lgid.gid_guid, lgid.gid_guid);
363                                 (void) ibt_cm_delay(IBT_CM_DELAY_REQ,
364                                     evp->cm_session_id, 10000000 /* 10 sec */,
365                                     NULL, 0);
366                                 ret = rds_session_reinit(sp, lgid);
367                                 if (ret != 0) {
368                                         rds_session_fini(sp);
369                                         sp->session_state =
370                                             RDS_SESSION_STATE_FAILED;
371                                         sp->session_failover = 0;
372                                         RDS_DPRINTF3("rds_failover_session",
373                                             "SP(%p) State "
374                                             "RDS_SESSION_STATE_FAILED", sp);
375                                         rw_exit(&sp->session_lock);
376                                         return (IBT_CM_REJECT);
377                                 }
378                         }
379                         ep = &sp->session_dataep;
380                 } else {
381                         ep = &sp->session_ctrlep;
382                 }

384                 break;
385         default:
386                 RDS_DPRINTF2(LABEL, "ERROR: SP(%p) is in an unexpected "
387                     "state: %d", sp, sp->session_state);
388                 rw_exit(&sp->session_lock);
389                 return (IBT_CM_REJECT);
390         }
```

```
392         sp->session_failover = 0; /* reset any previous value */
393         if (cmp.cmp_failover) {
394                 RDS_DPRINTF2("rds_handle_cm_req",
395                     "SP(%p) Failover Session (BP %p)", sp, cmp.cmp_last_bufid);
396                 sp->session_failover = 1;
397         }

399         mutex_enter(&ep->ep_lock);
400         if (ep->ep_state == RDS_EP_STATE_UNCONNECTED) {
401                 ep->ep_state = RDS_EP_STATE_PASSIVE_PENDING;
402                 sp->session_type = RDS_SESSION_PASSIVE;
403                 rw_exit(&sp->session_lock);
404         } else if (ep->ep_state == RDS_EP_STATE_ACTIVE_PENDING) {
405                 rw_exit(&sp->session_lock);
406                 /*
407                  * Peer to peer connection. There is an active
408                  * connection pending on this ep. The one with
409                  * greater port guid becomes active and the
410                  * other becomes passive.
411                  */
412                 RDS_DPRINTF2("rds_handle_cm_req",
413                     "EP(%p) Peer-peer connection handling", ep);
414                 if (lgid.gid_guid > rgid.gid_guid) {
415                         /* this node is active so reject this request */
416                         mutex_exit(&ep->ep_lock);
417                         RDS_DPRINTF2(LABEL, "SP(%p) EP(%p): "
418                             "Rejecting passive in favor of active", sp, ep);
419                         return (IBT_CM_REJECT);
420                 } else {
421                         /*
422                          * This session is not the active end, change it
423                          * to passive end.
424                          */
425                         ep->ep_state = RDS_EP_STATE_PASSIVE_PENDING;

427                         rw_enter(&sp->session_lock, RW_WRITER);
428                         sp->session_type = RDS_SESSION_PASSIVE;
429                         sp->session_lgid = lgid;
430                         sp->session_rgid = rgid;
431                         rw_exit(&sp->session_lock);
432                 }
433         } else {
434                 rw_exit(&sp->session_lock);
435         }

437         ep->ep_lbufid = cmp.cmp_last_bufid;
438         ep->ep_ackwr.wr.rc.rcwr.rdma.rdma_raddr = (ib_vaddr_t)cmp.cmp_ack_addr;
439         ep->ep_ackwr.wr.rc.rcwr.rdma.rdma_rkey = cmp.cmp_ack_rkey;
440         cmp.cmp_last_bufid = ep->ep_rbufid;
441         cmp.cmp_ack_addr = ep->ep_ack_addr;
442         cmp.cmp_ack_rkey = ep->ep_ack_rkey;
443         mutex_exit(&ep->ep_lock);

445         /* continue with accepting the connection request for this channel */
446         chanhdl = rds_ep_alloc_rc_channel(ep, reqp->req_prim_hca_port);
447         if (chanhdl == NULL) {
448                 mutex_enter(&ep->ep_lock);
449                 ep->ep_state = RDS_EP_STATE_UNCONNECTED;
450                 mutex_exit(&ep->ep_lock);
451                 return (IBT_CM_REJECT);
452         }

454         /* pre-post recv buffers in the RQ */
455         rds_post_recv_buf((void *)chanhdl);
```



```
457         rargsp->cm_ret_len = sizeof (rds_cm_private_data_t);
458         bcopy((uint8_t *)&cmp, rcmp, sizeof (rds_cm_private_data_t));
459         rargsp->cm_ret.rep.cm_channel = chanhdl;
460         rargsp->cm_ret.rep.cm_rdma_ra_out = 4;
461         rargsp->cm_ret.rep.cm_rdma_ra_in = 4;
462         rargsp->cm_ret.rep.cm_rnr_retry_cnt = MinRnrRetry;

464         RDS_DPRINTF2("rds_handle_cm_req", "Return: SP(%p) EP(%p) Chan (%p)",
465             sp, ep, chanhdl);

467         return (IBT_CM_ACCEPT);
468 }
```

unchanged_portion_omitted_

```
**********************************************************
   63952 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/clients/rds/rdsib_ep.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 756 /* Free the session */
 757 static void
 758 rds_destroy_session(rds_session_t *sp)
 759 {
 760         rds_ep_t        *ep;
 761         rds_bufpool_t   *pool;

 763         ASSERT((sp->session_state == RDS_SESSION_STATE_CLOSED) ||
 764             (sp->session_state == RDS_SESSION_STATE_FAILED) ||
 765             (sp->session_state == RDS_SESSION_STATE_FINI) ||
 766             (sp->session_state == RDS_SESSION_STATE_PASSIVE_CLOSING));

 768         rw_enter(&sp->session_lock, RW_READER);
 769         RDS_DPRINTF2("rds_destroy_session", "SP(%p) State: %d", sp,
 770             sp->session_state);
 771         while (!((sp->session_state == RDS_SESSION_STATE_CLOSED) ||
 772             (sp->session_state == RDS_SESSION_STATE_FAILED) ||
 773             (sp->session_state == RDS_SESSION_STATE_FINI))) {
 774                 rw_exit(&sp->session_lock);
 775                 delay(drv_sectohz(1));
 775                 delay(drv_usectohz(1000000));
 776                 rw_enter(&sp->session_lock, RW_READER);
 777                 RDS_DPRINTF2("rds_destroy_session", "SP(%p) State: %d WAITING "
 778                     "ON SESSION", sp, sp->session_state);
 779         }
 780         rw_exit(&sp->session_lock);

 782         /* data channel */
 783         ep = &sp->session_dataep;

 785         /* send pool locks */
 786         pool = &ep->ep_sndpool;
 787         cv_destroy(&pool->pool_cv);
 788         mutex_destroy(&pool->pool_lock);

 790         /* recv pool locks */
 791         pool = &ep->ep_rcvpool;
 792         cv_destroy(&pool->pool_cv);
 793         mutex_destroy(&pool->pool_lock);
 794         mutex_destroy(&ep->ep_recvqp.qp_lock);

 796         /* control channel */
 797         ep = &sp->session_ctrlep;

 799         /* send pool locks */
 800         pool = &ep->ep_sndpool;
 801         cv_destroy(&pool->pool_cv);
 802         mutex_destroy(&pool->pool_lock);

 804         /* recv pool locks */
 805         pool = &ep->ep_rcvpool;
 806         cv_destroy(&pool->pool_cv);
 807         mutex_destroy(&pool->pool_lock);
 808         mutex_destroy(&ep->ep_recvqp.qp_lock);

 810         /* session */
 811         rw_destroy(&sp->session_lock);
 812         rw_destroy(&sp->session_local_portmap_lock);
 813         rw_destroy(&sp->session_remote_portmap_lock);
```



```
 815         /* free the session */
 816         kmem_free(sp, sizeof (rds_session_t));

 818         RDS_DPRINTF2("rds_destroy_session", "SP(%p) Return", sp);
 819 }

 821 /* This is called on the taskq thread */
 822 void
 823 rds_failover_session(void *arg)
 824 {
 825         rds_session_t   *sp = (rds_session_t *)arg;
 826         ib_gid_t        lgid, rgid;
 827         ipaddr_t        myip, remip;
 828         int             ret, cnt = 0;
 829         uint8_t         sp_state;

 831         RDS_DPRINTF2("rds_failover_session", "Enter: (%p)", sp);

 833         /* Make sure the session is still alive */
 834         if (rds_session_lkup_by_sp(sp) == B_FALSE) {
 835                 RDS_DPRINTF2("rds_failover_session",
 836                     "Return: SP(%p) not ALIVE", sp);
 837                 return;
 838         }

 840         RDS_INCR_FAILOVERS();

 842         rw_enter(&sp->session_lock, RW_WRITER);
 843         if (sp->session_type != RDS_SESSION_ACTIVE) {
 844                 /*
 845                  * The remote side must have seen the error and initiated
 846                  * a re-connect.
 847                  */
 848                 RDS_DPRINTF2("rds_failover_session",
 849                     "SP(%p) has become passive", sp);
 850                 rw_exit(&sp->session_lock);
 851                 return;
 852         }
 853         sp->session_failover = 1;
 854         sp_state = sp->session_state;
 855         rw_exit(&sp->session_lock);

 857         /*
 858          * The session is in ERROR state but close both channels
 859          * for a clean start.
 860          */
 861         if (sp_state == RDS_SESSION_STATE_ERROR) {
 862                 rds_session_close(sp, IBT_BLOCKING, 1);
 863         }

 865         /* wait 1 sec before re-connecting */
 866         delay(drv_sectohz(1));
 866         delay(drv_usectohz(1000000));

 868         do {
 869                 ibt_ip_path_attr_t      ipattr;
 870                 ibt_ip_addr_t           dstip;

 872                 /* The ipaddr should be in the network order */
 873                 myip = sp->session_myip;
 874                 remip = sp->session_remip;
 875                 ret = rds_sc_path_lookup(&myip, &remip);
 876                 if (ret == 0) {
 877                         RDS_DPRINTF2(LABEL, "Path not found (0x%x 0x%x)",
 878                             myip, remip);
```

```
879                 }
880                 /* check if we have (new) path from the source to destination */
881                 lgid.gid_prefix = 0;
882                 lgid.gid_guid = 0;
883                 rgid.gid_prefix = 0;
884                 rgid.gid_guid = 0;

886                 bzero(&ipattr, sizeof (ibt_ip_path_attr_t));
887                 dstip.family = AF_INET;
888                 dstip.un.ip4addr = remip;
889                 ipattr.ipa_dst_ip = &dstip;
890                 ipattr.ipa_src_ip.family = AF_INET;
891                 ipattr.ipa_src_ip.un.ip4addr = myip;
892                 ipattr.ipa_ndst = 1;
893                 ipattr.ipa_max_paths = 1;
894                 RDS_DPRINTF2(LABEL, "ibt_get_ip_paths: 0x%x <-> 0x%x ",
895                     myip, remip);
896                 ret = ibt_get_ip_paths(rdsib_statep->rds_ibhdl,
897                     IBT_PATH_NO_FLAGS, &ipattr, &sp->session_pinfo, NULL, NULL);
898                 if (ret == IBT_SUCCESS) {
899                         RDS_DPRINTF2(LABEL, "ibt_get_ip_paths success");
900                         lgid = sp->session_pinfo.
901                             pi_prim_cep_path.cep_adds_vect.av_sgid;
902                         rgid = sp->session_pinfo.
903                             pi_prim_cep_path.cep_adds_vect.av_dgid;
904                         break;
905                 }

907                 RDS_DPRINTF2(LABEL, "ibt_get_ip_paths failed, ret: %d ", ret);

909                 /* wait 1 sec before re-trying */
910                 delay(drv_sectohz(1));
910                 delay(drv_usectohz(1000000));
911                 cnt++;
912         } while (cnt < 5);

914         if (ret != IBT_SUCCESS) {
915                 rw_enter(&sp->session_lock, RW_WRITER);
916                 if (sp->session_type == RDS_SESSION_ACTIVE) {
917                         rds_session_fini(sp);
918                         sp->session_state = RDS_SESSION_STATE_FAILED;
919                         sp->session_failover = 0;
920                         RDS_DPRINTF3("rds_failover_session",
921                             "SP(%p) State RDS_SESSION_STATE_FAILED", sp);
922                 } else {
923                         RDS_DPRINTF2("rds_failover_session",
924                             "SP(%p) has become passive", sp);
925                 }
926                 rw_exit(&sp->session_lock);
927                 return;
928         }

930         RDS_DPRINTF2(LABEL, "lgid: %llx:%llx rgid: %llx:%llx",
931             lgid.gid_prefix, lgid.gid_guid, rgid.gid_prefix,
932             rgid.gid_guid);

934         rw_enter(&sp->session_lock, RW_WRITER);
935         if (sp->session_type != RDS_SESSION_ACTIVE) {
936                 /*
937                  * The remote side must have seen the error and initiated
938                  * a re-connect.
939                  */
940                 RDS_DPRINTF2("rds_failover_session",
941                     "SP(%p) has become passive", sp);
942                 rw_exit(&sp->session_lock);
943                 return;
```



```
944         }

946         /* move the session to init state */
947         ret = rds_session_reinit(sp, lgid);
948         sp->session_lgid = lgid;
949         sp->session_rgid = rgid;
950         if (ret != 0) {
951                 rds_session_fini(sp);
952                 sp->session_state = RDS_SESSION_STATE_FAILED;
953                 sp->session_failover = 0;
954                 RDS_DPRINTF3("rds_failover_session",
955                     "SP(%p) State RDS_SESSION_STATE_FAILED", sp);
956                 rw_exit(&sp->session_lock);
957                 return;
958         } else {
959                 sp->session_state = RDS_SESSION_STATE_INIT;
960                 RDS_DPRINTF3("rds_failover_session",
961                     "SP(%p) State RDS_SESSION_STATE_INIT", sp);
962         }
963         rw_exit(&sp->session_lock);

965         rds_session_open(sp);

967         RDS_DPRINTF2("rds_failover_session", "Return: (%p)", sp);
968 }
_____unchanged_portion_omitted_

1099 /*
1100  * Can be called:
1101  * 1. on driver detach
1102  * 2. on taskq thread
1103  * arg is always NULL
1104  */
1105 /* ARGSUSED */
1106 void
1107 rds_close_sessions(void *arg)
1108 {
1109         rds_session_t *sp, *spnextp;

1111         RDS_DPRINTF2("rds_close_sessions", "Enter");

1113         /* wait until all the buffers are freed by the sockets */
1114         while (RDS_GET_RXPKTS_PEND() != 0) {
1115                 /* wait one second and try again */
1116                 RDS_DPRINTF2("rds_close_sessions", "waiting on "
1117                     "pending packets", RDS_GET_RXPKTS_PEND());
1118                 delay(drv_sectohz(1));
1118                 delay(drv_usectohz(1000000));
1119         }
1120         RDS_DPRINTF2("rds_close_sessions", "No more RX packets pending");

1122         /* close all the sessions */
1123         rw_enter(&rdsib_statep->rds_sessionlock, RW_WRITER);
1124         sp = rdsib_statep->rds_sessionlistp;
1125         while (sp) {
1126                 rw_enter(&sp->session_lock, RW_WRITER);
1127                 RDS_DPRINTF2("rds_close_sessions", "SP(%p) State: %d", sp,
1128                     sp->session_state);
1129                 rds_close_this_session(sp, 2);
1130                 rw_exit(&sp->session_lock);
1131                 sp = sp->session_nextp;
1132         }

1134         sp = rdsib_statep->rds_sessionlistp;
1135         rdsib_statep->rds_sessionlistp = NULL;
1136         rdsib_statep->rds_nsessions = 0;
```

```
1137         rw_exit(&rdsib_statep->rds_sessionlock);

1139         while (sp) {
1140                 spnextp = sp->session_nextp;
1141                 rds_destroy_session(sp);
1142                 RDS_DECR_SESS();
1143                 sp = spnextp;
1144         }

1146         /* free the global pool */
1147         rds_free_recv_caches(rdsib_statep);

1149         RDS_DPRINTF2("rds_close_sessions", "Return");
1150 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   48565 Wed Aug 19 07:25:05 2015
new/usr/src/uts/common/io/ib/clients/rds/rdsib_ib.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1527 void rds_close_this_session(rds_session_t *sp, uint8_t wait);
1528 int rds_post_control_message(rds_session_t *sp, uint8_t code, in_port_t port);

1530 static void
1531 rdsib_del_hca(rds_state_t *statep, ib_guid_t hca_guid)
1532 {
1533         rds_session_t   *sp;
1534         rds_hca_t       *hcap;
1535         rds_hca_state_t saved_state;
1536         int             ret, ix;

1538         RDS_DPRINTF2("rdsib_del_hca", "Enter: GUID: 0x%llx", hca_guid);

1540         /*
1541          * This should be a write lock as we don't want anyone to get access
1542          * to the hcap while we are modifing its contents
1543          */
1544         rw_enter(&statep->rds_hca_lock, RW_WRITER);

1546         hcap = statep->rds_hcalistp;
1547         while ((hcap != NULL) && (hcap->hca_guid != hca_guid)) {
1548                 hcap = hcap->hca_nextp;
1549         }

1551         /* Prevent initiating any new activity on this HCA */
1552         ASSERT(hcap != NULL);
1553         saved_state = hcap->hca_state;
1554         hcap->hca_state = RDS_HCA_STATE_STOPPING;

1556         rw_exit(&statep->rds_hca_lock);

1558         /*
1559          * stop the outgoing traffic and close any active sessions on this hca.
1560          * Any pending messages in the SQ will be allowed to complete.
1561          */
1562         rw_enter(&statep->rds_sessionlock, RW_READER);
1563         sp = statep->rds_sessionlistp;
1564         while (sp) {
1565                 if (sp->session_hca_guid != hca_guid) {
1566                         sp = sp->session_nextp;
1567                         continue;
1568                 }

1570                 rw_enter(&sp->session_lock, RW_WRITER);
1571                 RDS_DPRINTF2("rdsib_del_hca", "SP(%p) State: %d", sp,
1572                     sp->session_state);
1573                 /*
1574                  * We are changing the session state in advance. This prevents
1575                  * further messages to be posted to the SQ. We then
1576                  * send a control message to the remote and tell it close
1577                  * the session.
1578                  */
1579                 sp->session_state = RDS_SESSION_STATE_HCA_CLOSING;
1580                 RDS_DPRINTF3("rds_handle_cm_conn_closed", "SP(%p) State "
1581                     "RDS_SESSION_STATE_PASSIVE_CLOSING", sp);
1582                 rw_exit(&sp->session_lock);

1584                 /*
1585                  * wait until the sendq is empty then tell the remote to
```



```
1586                  * close this session. This enables for graceful shutdown of
1587                  * the session
1588                  */
1589                 (void) rds_is_sendq_empty(&sp->session_dataep, 2);
1590                 (void) rds_post_control_message(sp,
1591                     RDS_CTRL_CODE_CLOSE_SESSION, 0);

1593                 sp = sp->session_nextp;
1594         }

1596         /* wait until all the sessions are off this HCA */
1597         sp = statep->rds_sessionlistp;
1598         while (sp) {
1599                 if (sp->session_hca_guid != hca_guid) {
1600                         sp = sp->session_nextp;
1601                         continue;
1602                 }

1604                 rw_enter(&sp->session_lock, RW_READER);
1605                 RDS_DPRINTF2("rdsib_del_hca", "SP(%p) State: %d", sp,
1606                     sp->session_state);

1608                 while ((sp->session_state == RDS_SESSION_STATE_HCA_CLOSING) ||
1609                     (sp->session_state == RDS_SESSION_STATE_ERROR) ||
1610                     (sp->session_state == RDS_SESSION_STATE_PASSIVE_CLOSING) ||
1611                     (sp->session_state == RDS_SESSION_STATE_CLOSED)) {
1612                         rw_exit(&sp->session_lock);
1613                         delay(drv_sectohz(1));
1613                         delay(drv_usectohz(1000000));
1614                         rw_enter(&sp->session_lock, RW_READER);
1615                         RDS_DPRINTF2("rdsib_del_hca", "SP(%p) State: %d", sp,
1616                             sp->session_state);
1617                 }

1619                 rw_exit(&sp->session_lock);

1621                 sp = sp->session_nextp;
1622         }
1623         rw_exit(&statep->rds_sessionlock);

1625         /*
1626          * if rdsib_close_ib was called before this, then that would have
1627          * unbound the service on all ports. In that case, the HCA structs
1628          * will contain stale bindhdls. Hence, we do not call unbind unless
1629          * the service is still registered.
1630          */
1631         if (statep->rds_srvhdl != NULL) {
1632                 /* unbind RDS service on all ports on this HCA */
1633                 for (ix = 0; ix < hcap->hca_nports; ix++) {
1634                         if (hcap->hca_bindhdl[ix] == NULL) {
1635                                 continue;
1636                         }

1638                         RDS_DPRINTF2("rdsib_del_hca",
1639                             "Unbinding Service: port: %d, bindhdl: %p",
1640                             ix + 1, hcap->hca_bindhdl[ix]);
1641                         (void) ibt_unbind_service(rdsib_statep->rds_srvhdl,
1642                             hcap->hca_bindhdl[ix]);
1643                         hcap->hca_bindhdl[ix] = NULL;
1644                 }
1645         }

1647         RDS_DPRINTF2("rdsib_del_hca", "HCA(%p) State: %d", hcap,
1648             hcap->hca_state);

1650         switch (saved_state) {
```

```
1651         case RDS_HCA_STATE_MEM_REGISTERED:
1652                 ASSERT(hcap->hca_mrhdl != NULL);
1653                 ret = ibt_deregister_mr(hcap->hca_hdl, hcap->hca_mrhdl);
1654                 if (ret != IBT_SUCCESS) {
1655                         RDS_DPRINTF2("rdsib_del_hca",
1656                             "ibt_deregister_mr failed: %d", ret);
1657                         return;
1658                 }
1659                 hcap->hca_mrhdl = NULL;
1660                 /* FALLTHRU */
1661         case RDS_HCA_STATE_OPEN:
1662                 ASSERT(hcap->hca_hdl != NULL);
1663                 ASSERT(hcap->hca_pdhdl != NULL);


1666                 ret = ibt_free_pd(hcap->hca_hdl, hcap->hca_pdhdl);
1667                 if (ret != IBT_SUCCESS) {
1668                         RDS_DPRINTF2("rdsib_del_hca",
1669                             "ibt_free_pd failed: %d", ret);
1670                 }

1672                 (void) ibt_free_portinfo(hcap->hca_pinfop, hcap->hca_pinfo_sz);

1674                 ret = ibt_close_hca(hcap->hca_hdl);
1675                 if (ret != IBT_SUCCESS) {
1676                         RDS_DPRINTF2("rdsib_del_hca",
1677                             "ibt_close_hca failed: %d", ret);
1678                 }

1680                 hcap->hca_hdl = NULL;
1681                 hcap->hca_pdhdl = NULL;
1682                 hcap->hca_lkey = 0;
1683                 hcap->hca_rkey = 0;
1684         }

1686         /*
1687          * This should be a write lock as we don't want anyone to get access
1688          * to the hcap while we are modifing its contents
1689          */
1690         rw_enter(&statep->rds_hca_lock, RW_WRITER);
1691         hcap->hca_state = RDS_HCA_STATE_REMOVED;
1692         rw_exit(&statep->rds_hca_lock);

1694         RDS_DPRINTF2("rdsib_del_hca", "Return: GUID: 0x%llx", hca_guid);
1695 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26469 Wed Aug 19 07:25:06 2015
new/usr/src/uts/common/io/ib/clients/rdsv3/af_rds.c
XXXX introduce drv_sectohz
**********************************************************
```

```
   1 /*
   2  * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
   3  */

   5 /*
   6  * This file contains code imported from the OFED rds source file af_rds.c
   7  * Oracle elects to have and use the contents of af_rds.c under and governed
   8  * by the OpenIB.org BSD license (see below for full license text). However,
   9  * the following notice accompanied the original version of this file:
  10  */

  12 /*
  13  * Copyright (c) 2006 Oracle.  All rights reserved.
  14  *
  15  * This software is available to you under a choice of one of two
  16  * licenses.  You may choose to be licensed under the terms of the GNU
  17  * General Public License (GPL) Version 2, available from the file
  18  * COPYING in the main directory of this source tree, or the
  19  * OpenIB.org BSD license below:
  20  *
  21  *     Redistribution and use in source and binary forms, with or
  22  *     without modification, are permitted provided that the following
  23  *     conditions are met:
  24  *
  25  *      - Redistributions of source code must retain the above
  26  *        copyright notice, this list of conditions and the following
  27  *        disclaimer.
  28  *
  29  *      - Redistributions in binary form must reproduce the above
  30  *        copyright notice, this list of conditions and the following
  31  *        disclaimer in the documentation and/or other materials
  32  *        provided with the distribution.
  33  *
  34  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
  35  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
  36  * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
  37  * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
  38  * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
  39  * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
  40  * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
  41  * SOFTWARE.
  42  *
  43  */
  44 #include <sys/types.h>
  45 #include <sys/stat.h>
  46 #include <sys/conf.h>
  47 #include <sys/ddi.h>
  48 #include <sys/sunddi.h>
  49 #include <sys/modctl.h>
  50 #include <sys/rds.h>
  51 #include <sys/stropts.h>
  52 #include <sys/socket.h>
  53 #include <sys/socketvar.h>
  54 #include <sys/sockio.h>
  55 #include <sys/sysmacros.h>

  57 #include <inet/ip.h>
  58 #include <net/if_types.h>

  60 #include <sys/ib/clients/rdsv3/rdsv3.h>
  61 #include <sys/ib/clients/rdsv3/rdma.h>
```



```
  62 #include <sys/ib/clients/rdsv3/rdma_transport.h>
  63 #include <sys/ib/clients/rdsv3/rdsv3_debug.h>

  65 extern void rdsv3_remove_bound(struct rdsv3_sock *rds);
  66 extern int rdsv3_verify_bind_address(ipaddr_t addr);

  68 extern ddi_taskq_t      *rdsv3_taskq;
  69 extern struct rdma_cm_id *rdsv3_rdma_listen_id;

  71 /* this is just used for stats gathering :/ */
  72 kmutex_t rdsv3_sock_lock;
  73 static unsigned long rdsv3_sock_count;
  74 list_t rdsv3_sock_list;

  76 /*
  77  * This is called as the final descriptor referencing this socket is closed.
  78  * We have to unbind the socket so that another socket can be bound to the
  79  * address it was using.
  80  *
  81  * We have to be careful about racing with the incoming path.  sock_orphan()
  82  * sets SOCK_DEAD and we use that as an indicator to the rx path that new
  83  * messages shouldn't be queued.
  84  */
  85 /* ARGSUSED */
  86 static int
  87 rdsv3_release(sock_lower_handle_t proto_handle, int flgs, cred_t *cr)
  88 {
  89         struct rsock *sk = (struct rsock *)proto_handle;
  90         struct rdsv3_sock *rs;

  92         if (!sk)
  93                 goto out;

  95         rs = rdsv3_sk_to_rs(sk);
  96         RDSV3_DPRINTF4("rdsv3_release", "Enter(rs: %p, sk: %p)", rs, sk);

  98         rdsv3_sk_sock_orphan(sk);
  99         rdsv3_cong_remove_socket(rs);
 100         rdsv3_remove_bound(rs);

 102         /*
 103          * Note - rdsv3_clear_recv_queue grabs rs_recv_lock, so
 104          * that ensures the recv path has completed messing
 105          * with the socket.
 106          *
 107          * Note2 - rdsv3_clear_recv_queue(rs) should be called first
 108          * to prevent some race conditions, which is different from
 109          * the Linux code.
 110          */
 111         rdsv3_clear_recv_queue(rs);
 112         rdsv3_send_drop_to(rs, NULL);
 113         rdsv3_rdma_drop_keys(rs);
 114         (void) rdsv3_notify_queue_get(rs, NULL);

 116         mutex_enter(&rdsv3_sock_lock);
 117         list_remove_node(&rs->rs_item);
 118         rdsv3_sock_count--;
 119         mutex_exit(&rdsv3_sock_lock);

 121         while (sk->sk_refcount > 1) {
 122                 /* wait for 1 sec and try again */
 123                 delay(drv_sectohz(1));
 123                 delay(drv_usectohz(1000000));
 124         }

 126         /* this will free the rs and sk */
```

```
 127         rdsv3_sk_sock_put(sk);

 129         RDSV3_DPRINTF4("rdsv3_release", "Return (rds: %p)", rs);
 130 out:
 131         return (0);
 132 }
```

unchanged_portion_omitted_

```
**********************************************************
   31434 Wed Aug 19 07:25:06 2015
new/usr/src/uts/common/io/ib/clients/rdsv3/rdsv3_impl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 548 /* XXX */
 549 void
 550 rdsv3_flush_workqueue(rdsv3_workqueue_struct_t *wq)
 551 {
 552         RDSV3_DPRINTF4("rdsv3_flush_workqueue", "Enter(wq: %p)", wq);

 554         mutex_enter(&wq->wq_lock);
 555         switch (wq->wq_state) {
 556         case RDSV3_WQ_THREAD_IDLE:
 557                 /* nothing to do */
 558                 ASSERT(list_is_empty(&wq->wq_queue));
 559                 break;

 561         case RDSV3_WQ_THREAD_RUNNING:
 562                 wq->wq_state = RDSV3_WQ_THREAD_FLUSHING;
 563                 /* FALLTHRU */
 564         case RDSV3_WQ_THREAD_FLUSHING:
 565                 /* already flushing, wait until the flushing is complete */
 566                 do {
 567                         mutex_exit(&wq->wq_lock);
 568                         delay(drv_sectohz(1));
 568                         delay(drv_usectohz(1000000));
 569                         mutex_enter(&wq->wq_lock);
 570                 } while (wq->wq_state == RDSV3_WQ_THREAD_FLUSHING);
 571                 break;
 572         case RDSV3_WQ_THREAD_EXITING:
 573                 mutex_exit(&wq->wq_lock);
 574                 rdsv3_worker_thread(wq);
 575                 return;
 576         }
 577         mutex_exit(&wq->wq_lock);

 579         RDSV3_DPRINTF4("rdsv3_flush_workqueue", "Return(wq: %p)", wq);
 580 }

 582 void
 583 rdsv3_queue_work(rdsv3_workqueue_struct_t *wq, rdsv3_work_t *wp)
 584 {
 585         RDSV3_DPRINTF4("rdsv3_queue_work", "Enter(wq: %p, wp: %p)", wq, wp);

 587         mutex_enter(&wq->wq_lock);

 589         if (list_link_active(&wp->work_item)) {
 590                 /* This is already in the queue, ignore this call */
 591                 mutex_exit(&wq->wq_lock);
 592                 RDSV3_DPRINTF3("rdsv3_queue_work", "already queued: %p", wp);
 593                 return;
 594         }

 596         switch (wq->wq_state) {
 597         case RDSV3_WQ_THREAD_RUNNING:
 598                 list_insert_tail(&wq->wq_queue, wp);
 599                 mutex_exit(&wq->wq_lock);
 600                 break;

 602         case RDSV3_WQ_THREAD_FLUSHING:
 603                 do {
 604                         mutex_exit(&wq->wq_lock);
 605                         delay(drv_sectohz(1));
```



```
 605                         delay(drv_usectohz(1000000));
 606                         mutex_enter(&wq->wq_lock);
 607                 } while (wq->wq_state == RDSV3_WQ_THREAD_FLUSHING);

 609                 if (wq->wq_state == RDSV3_WQ_THREAD_RUNNING) {
 610                         list_insert_tail(&wq->wq_queue, wp);
 611                         mutex_exit(&wq->wq_lock);
 612                         break;
 613                 }
 614                 /* FALLTHRU */

 616         case RDSV3_WQ_THREAD_IDLE:
 617                 list_insert_tail(&wq->wq_queue, wp);
 618                 wq->wq_state = RDSV3_WQ_THREAD_RUNNING;
 619                 mutex_exit(&wq->wq_lock);

 621                 (void) ddi_taskq_dispatch(rdsv3_taskq, rdsv3_worker_thread, wq,
 622                     DDI_SLEEP);
 623                 break;

 625         case RDSV3_WQ_THREAD_EXITING:
 626                 mutex_exit(&wq->wq_lock);
 627                 break;
 628         }

 630         RDSV3_DPRINTF4("rdsv3_queue_work", "Return(wq: %p, wp: %p)", wq, wp);
 631 }
______unchanged_portion_omitted_

 727 void
 728 rdsv3_destroy_task_workqueue(rdsv3_workqueue_struct_t *wq)
 729 {
 730         RDSV3_DPRINTF2("rdsv3_destroy_workqueue", "Enter");

 732         ASSERT(wq);

 734         mutex_enter(&wq->wq_lock);
 735         wq->wq_state = RDSV3_WQ_THREAD_EXITING;

 737         while (wq->wq_pending > 0) {
 738                 mutex_exit(&wq->wq_lock);
 739                 delay(drv_sectohz(1));
 739                 delay(drv_usectohz(1000000));
 740                 mutex_enter(&wq->wq_lock);
 741         };
 742         mutex_exit(&wq->wq_lock);

 744         rdsv3_flush_workqueue(wq);

 746         list_destroy(&wq->wq_queue);
 747         mutex_destroy(&wq->wq_lock);
 748         kmem_free(wq, sizeof (rdsv3_workqueue_struct_t));

 750         ASSERT(rdsv3_taskq);
 751         ddi_taskq_destroy(rdsv3_taskq);

 753         wq = NULL;
 754         rdsv3_taskq = NULL;

 756         RDSV3_DPRINTF2("rdsv3_destroy_workqueue", "Return");
 757 }
______unchanged_portion_omitted_

 766 #define RDSV3_NUM_TASKQ_THREADS 1
 767 rdsv3_workqueue_struct_t *
 768 rdsv3_create_task_workqueue(char *name)
```

```
 769 {
 770         rdsv3_workqueue_struct_t        *wq;

 772         RDSV3_DPRINTF2("create_singlethread_workqueue", "Enter (dip: %p)",
 773             rdsv3_dev_info);

 775         rdsv3_taskq = ddi_taskq_create(rdsv3_dev_info, name,
 776             RDSV3_NUM_TASKQ_THREADS, TASKQ_DEFAULTPRI, 0);
 777         if (rdsv3_taskq == NULL) {
 778                 RDSV3_DPRINTF2(__FILE__,
 779                     "ddi_taskq_create failed for rdsv3_taskq");
 780                 return (NULL);
 781         }

 783         wq = kmem_zalloc(sizeof (rdsv3_workqueue_struct_t), KM_NOSLEEP);
 784         if (wq == NULL) {
 785                 RDSV3_DPRINTF2(__FILE__, "kmem_zalloc failed for wq");
 786                 ddi_taskq_destroy(rdsv3_taskq);
 787                 return (NULL);
 788         }

 790         list_create(&wq->wq_queue, sizeof (struct rdsv3_work_s),
 791             offsetof(struct rdsv3_work_s, work_item));
 792         mutex_init(&wq->wq_lock, NULL, MUTEX_DRIVER, NULL);
 793         wq->wq_state = RDSV3_WQ_THREAD_IDLE;
 794         wq->wq_pending = 0;
 795         rdsv3_one_sec_in_hz = drv_sectohz(1);
 795         rdsv3_one_sec_in_hz = drv_usectohz(1000000);

 797         RDSV3_DPRINTF2("create_singlethread_workqueue", "Return");

 799         return (wq);
 800 }
______unchanged_portion_omitted_
```

```
**********************************************************
  135255 Wed Aug 19 07:25:06 2015
new/usr/src/uts/common/io/ib/mgt/ibcm/ibcm_path.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 188 /*ARGSUSED*/
 189 static void
 190 ibcm_path_cache_timeout_cb(void *arg)
 191 {
 192         clock_t timeout_in_hz;

 194         timeout_in_hz = drv_sectohz(ibcm_path_cache_timeout);
 194         timeout_in_hz = drv_usectohz(ibcm_path_cache_timeout * 1000000);
 195         mutex_enter(&ibcm_path_cache_mutex);
 196         ibcm_path_cache_invalidate = 1; /* invalidate cache on next check */
 197         if (ibcm_path_cache_timeout_id)
 198                 ibcm_path_cache_timeout_id = timeout(ibcm_path_cache_timeout_cb,
 199                     NULL, timeout_in_hz);
 200         /* else we're in _fini */
 201         mutex_exit(&ibcm_path_cache_mutex);
 202 }

 204 void
 205 ibcm_path_cache_init(void)
 206 {
 207         clock_t timeout_in_hz;
 208         int cache_size = ibcm_path_cache_size_init;
 209         ibcm_path_cache_t *path_cachep;

 211         timeout_in_hz = drv_sectohz(ibcm_path_cache_timeout);
 211         timeout_in_hz = drv_usectohz(ibcm_path_cache_timeout * 1000000);
 212         path_cachep = kmem_zalloc(cache_size * sizeof (*path_cachep), KM_SLEEP);
 213         mutex_init(&ibcm_path_cache_mutex, NULL, MUTEX_DEFAULT, NULL);
 214         mutex_enter(&ibcm_path_cache_mutex);
 215         ibcm_path_cache_size = cache_size;
 216         ibcm_path_cachep = path_cachep;
 217         ibcm_path_cache_timeout_id = timeout(ibcm_path_cache_timeout_cb,
 218             NULL, timeout_in_hz);
 219         mutex_exit(&ibcm_path_cache_mutex);
 220 }
______unchanged_portion_omitted_
```

```
**********************************************************
   41435 Wed Aug 19 07:25:06 2015
new/usr/src/uts/common/io/ib/mgt/ibmf/ibmf.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 223 /* ARGSUSED */
 224 int
 225 ibmf_unregister(ibmf_handle_t *ibmf_handlep, uint_t flags)
 226 {
 227         ibmf_ci_t       *cip;
 228         ibmf_client_t   *clientp;
 229         boolean_t       error = B_FALSE;
 230         int             status = IBMF_SUCCESS;
 231         char            errmsg[128];
 232         int             secs;

 234         clientp = (ibmf_client_t *)*ibmf_handlep;

 236         IBMF_TRACE_2(IBMF_TNF_DEBUG, DPRINT_L4, ibmf_unregister_start,
 237             IBMF_TNF_TRACE, "", "ibmf_unregister() enter, "
 238             "ibmf_handlep = %p, flags = 0x%x\n",
 239             tnf_opaque, ibmf_handle, *ibmf_handlep, tnf_uint, flags, flags);

 241         /* check for null ibmf_handlep */
 242         if (ibmf_handlep == NULL) {
 243                 (void) sprintf(errmsg,
 244                     "invalid argument, NULL pointer argument");
 245                 error = B_TRUE;
 246                 status = IBMF_INVALID_ARG;
 247                 goto bail;
 248         }

 250         /* validate ibmf_handlep */
 251         if (ibmf_i_is_ibmf_handle_valid(*ibmf_handlep) != IBMF_SUCCESS) {
 252                 (void) sprintf(errmsg, "bad ibmf registration handle");
 253                 error = B_TRUE;
 254                 status = IBMF_BAD_HANDLE;
 255                 goto bail;
 256         }

 258         /* check signature */
 259         if (IBMF_VERIFY_CLIENT_SIGNATURE(clientp) == B_FALSE) {
 260                 (void) sprintf(errmsg, "bad client signature");
 261                 error = B_TRUE;
 262                 status = IBMF_BAD_HANDLE;
 263                 goto bail;
 264         }

 266         /*
 267          * Verify the client does not have a receive callback registered.
 268          * If there are messages, give some time for the messages to be
 269          * cleaned up.
 270          */
 271         secs = 60;
 272         mutex_enter(&clientp->ic_mutex);
 273         while (clientp->ic_recv_cb == NULL && clientp->ic_msgs_alloced != 0 &&
 274             secs > 0) {
 275                 mutex_exit(&clientp->ic_mutex);
 276                 delay(drv_sectohz(1)); /* one second delay */
 276                 delay(drv_usectohz(1000000)); /* one second delay */
 277                 secs--;
 278                 mutex_enter(&clientp->ic_mutex);
 279         }
```



```
 281         if (clientp->ic_recv_cb != NULL || clientp->ic_msgs_alloced != 0) {
 282                 IBMF_TRACE_4(IBMF_TNF_NODEBUG, DPRINT_L1,
 283                     ibmf_unregister_err, IBMF_TNF_ERROR, "",
 284                     "ibmf_unregister(): %s, flags = 0x%x, recv_cb = 0x%p, "
 285                     "msgs_alloced = %d\n",
 286                     tnf_string, msg, "busy with resources", tnf_uint, ic_flags,
 287                     clientp->ic_flags, tnf_opaque, recv_cb, clientp->ic_recv_cb,
 288                     tnf_uint, msgs_allocd, clientp->ic_msgs_alloced);
 289                 IBMF_TRACE_0(IBMF_TNF_DEBUG, DPRINT_L4, ibmf_unregister_end,
 290                     IBMF_TNF_TRACE, "", "ibmf_unregister() exit\n");
 291                 mutex_exit(&clientp->ic_mutex);
 292                 return (IBMF_BUSY);
 293         }

 295         mutex_exit(&clientp->ic_mutex);

 297         cip = clientp->ic_myci;

 299         /* remove the client from the list of clients */
 300         ibmf_i_delete_client(cip, clientp);

 302         /* release the reference to the qp */
 303         ibmf_i_release_qp(cip, &clientp->ic_qp);

 305         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*clientp))

 307         /* and free the client structure */
 308         ibmf_i_free_client(clientp);

 310         /* release the ci; this may delete & free the ci structure */
 311         ibmf_i_release_ci(cip);

 313         /* decrement kstats for number of registered clients */
 314         mutex_enter(&cip->ci_mutex);
 315         IBMF_SUB32_PORT_KSTATS(cip, clients_registered, 1);
 316         mutex_exit(&cip->ci_mutex);

 318         *ibmf_handlep = NULL;

 320 bail:
 321         if (error) {
 322                 IBMF_TRACE_1(IBMF_TNF_NODEBUG, DPRINT_L1,
 323                     ibmf_unregister_err, IBMF_TNF_ERROR, "",
 324                     "ibmf_unregister(): %s\n", tnf_string, msg, errmsg);
 325         }

 327         IBMF_TRACE_0(IBMF_TNF_DEBUG, DPRINT_L4, ibmf_unregister_end,
 328             IBMF_TNF_TRACE, "", "ibmf_unregister() exit\n");

 330         return (status);
 331 }
______unchanged_portion_omitted_
```

```
**********************************************************
  118319 Wed Aug 19 07:25:06 2015
new/usr/src/uts/common/io/ib/mgt/ibmf/ibmf_saa_impl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1333 /*
1334  * ibmf_saa_impl_send_request:
1335  * Sends a request to the sa.  Can be used for both classportinfo and record
1336  * requests.  Will set up all data structures for using the multi-packet
1337  * protocol, create the mad, and send it.  Returns SA_SUCCESS if msg transport
1338  * worked, meaning succesful send for the async case and a succesful send and
1339  * recv for the sync case.
1340  */
1341 int
1342 ibmf_saa_impl_send_request(saa_impl_trans_info_t *trans_info)
1343 {
1344         uint16_t                attr_id;
1345         saa_client_data_t       *client_data;
1346         saa_port_t              *saa_portp;
1347         uint32_t                transport_flags;
1348         ibmf_msg_cb_t           ibmf_callback;
1349         void                    *ibmf_callback_arg;
1350         ibmf_msg_t              *msgp;
1351         ibmf_retrans_t          ibmf_retrans;
1352         uint16_t                sa_cap_mask;
1353         boolean_t               sleep_flag;
1354         int                     ibmf_status = IBMF_SUCCESS;
1355         int                     retry_count;
1356         uint16_t                mad_status;
1357         boolean_t               sa_is_redirected = B_FALSE;

1359         IBMF_TRACE_0(IBMF_TNF_DEBUG, DPRINT_L4,
1360             ibmf_saa_impl_send_request_start,
1361             IBMF_TNF_TRACE, "", "ibmf_saa_impl_send_request() enter\n");

1363         attr_id = trans_info->si_trans_attr_id;
1364         client_data = trans_info->si_trans_client_data;
1365         saa_portp   = trans_info->si_trans_port;

1367         /*
1368          * don't send on invalid entry
1369          * Note that there is a window where it could become
1370          * invalid after this test is done, but we'd rely on ibmf errors...
1371          */
1372         if (ibmf_saa_is_valid(saa_portp, B_FALSE) == B_FALSE) {

1374                 IBMF_TRACE_4(IBMF_TNF_NODEBUG, DPRINT_L1,
1375                     ibmf_saa_impl_send_request,
1376                     IBMF_TNF_ERROR, "",
1377                     "ibmf_saa_impl_send_request: %s, hca_guid = %016"
1378                     PRIx64 ", port_guid = %016" PRIx64
1379                     ", number = %d\n",
1380                     tnf_string, msg, "sending on invalid port",
1381                     tnf_opaque, hca_guid,
1382                     saa_portp->saa_pt_ibmf_reginfo.ir_ci_guid,
1383                     tnf_opaque, port_guid,
1384                     saa_portp->saa_pt_port_guid,
1385                     tnf_uint,   port,
1386                     saa_portp->saa_pt_ibmf_reginfo.ir_port_num);

1388                 ibmf_status = IBMF_REQ_INVALID;
1389                 goto bail;
1390         }
```



```
1392         /* check whether SA supports this attribute */
1393         mutex_enter(&saa_portp->saa_pt_mutex);

1395         sa_cap_mask = saa_portp->saa_pt_sa_cap_mask;
1396         sa_is_redirected = saa_portp->saa_pt_redirect_active;

1398         mutex_exit(&saa_portp->saa_pt_mutex);

1400         ibmf_status = ibmf_saa_impl_check_sa_support(sa_cap_mask, attr_id);

1402         if (ibmf_status != IBMF_SUCCESS) {

1404                 IBMF_TRACE_2(IBMF_TNF_NODEBUG, DPRINT_L1,
1405                     ibmf_saa_impl_send_request_err, IBMF_TNF_ERROR, "",
1406                     "ibmf_saa_impl_send_request: %s, ibmf_status = %d\n",
1407                     tnf_string, msg, "SA does not support attribute",
1408                     tnf_int, ibmf_status, ibmf_status);

1410                 goto bail;
1411         }

1413         /* make only non-blocking calls if this is an async request */
1414         if ((trans_info->si_trans_callback == NULL) &&
1415             (trans_info->si_trans_sub_callback == NULL)) {
1416                 ibmf_callback = NULL;
1417                 ibmf_callback_arg = NULL;
1418                 sleep_flag = B_TRUE;
1419         } else {
1420                 ibmf_callback = ibmf_saa_async_cb;
1421                 ibmf_callback_arg = (void *)trans_info;
1422                 sleep_flag = B_FALSE;
1423         }

1425         ibmf_status = ibmf_saa_impl_init_msg(trans_info, sleep_flag, &msgp,
1426             &transport_flags, &ibmf_retrans);
1427         if (ibmf_status != IBMF_SUCCESS) {

1429                 IBMF_TRACE_2(IBMF_TNF_NODEBUG, DPRINT_L1,
1430                     ibmf_saa_impl_send_request_err, IBMF_TNF_ERROR, "",
1431                     "ibmf_saa_impl_send_request: %s, ibmf_status = %d\n",
1432                     tnf_string, msg, "init_msg() failed",
1433                     tnf_int, ibmf_status, ibmf_status);

1435                 goto bail;
1436         }

1438         mutex_enter(&saa_portp->saa_pt_mutex);

1440         saa_portp->saa_pt_num_outstanding_trans++;

1442         mutex_exit(&saa_portp->saa_pt_mutex);

1444         /*
1445          * increment the number of outstanding transaction so
1446          * ibmf_close_sa_session() will wait.  classportinfo requests
1447          * don't have associated clients so check for valid clientp
1448          */
1449         if (client_data != NULL) {

1451                 mutex_enter(&client_data->saa_client_mutex);

1453                 client_data->saa_client_num_pending_trans++;

1455                 mutex_exit(&client_data->saa_client_mutex);
1456         }
```

```
1458         /*
1459          * make the call to msg_transport.  If synchronous and success,
1460          * check that the response mad isn't status busy.  If so, repeat the
1461          * call
1462          */
1463         retry_count = 0;

1465         /*
1466          * set the send time here. We only set this once at the beginning of
1467          * the transaction.  Retrying because of busys or mastersmlid changes
1468          * does not change the original send time.  It is meant to be an
1469          * absolute time out value and will only be used if there are other
1470          * problems (i.e. a buggy SA)
1471          */
1472         trans_info->si_trans_send_time = gethrtime();

1474         for (;;) {

1476                 ibmf_status = ibmf_msg_transport(saa_portp->saa_pt_ibmf_handle,
1477                     saa_portp->saa_pt_qp_handle, msgp, &ibmf_retrans,
1478                     ibmf_callback, ibmf_callback_arg, transport_flags);

1480                 if (ibmf_callback != NULL)
1481                         break;

1483                 /*
1484                  * stop here for non-sequenced transactions since they wouldn't
1485                  * receive a timeout or busy response
1486                  */
1487                 if (!(transport_flags & IBMF_MSG_TRANS_FLAG_SEQ))
1488                         break;

1490                 /*
1491                  * if the transaction timed out and this was a synchronous
1492                  * request there's a possiblity we were talking to the wrong
1493                  * master smlid or that the SA has stopped responding on the
1494                  * redirected desination (if redirect is active).
1495                  * Check this and retry if necessary.
1496                  */
1497                 if ((ibmf_status == IBMF_TRANS_TIMEOUT) &&
1498                     (sleep_flag == B_TRUE)) {
1499                         if (sa_is_redirected == B_TRUE) {
1500                                 ibmf_status = ibmf_saa_impl_revert_to_qp1(
1501                                     saa_portp, msgp, ibmf_callback,
1502                                     ibmf_callback_arg, transport_flags);
1503                         } else {
1504                                 ibmf_status = ibmf_saa_impl_new_smlid_retry(
1505                                     saa_portp, msgp, ibmf_callback,
1506                                     ibmf_callback_arg, transport_flags);
1507                         }
1508                 }

1510                 /*
1511                  * if the transaction timed out (and retrying with a new SM LID
1512                  * didn't help) check how long it's been since we received an SA
1513                  * packet.  If it hasn't been max_wait_time then retry the
1514                  * request.
1515                  */
1516                 if ((ibmf_status == IBMF_TRANS_TIMEOUT) &&
1517                     (sleep_flag == B_TRUE)) {

1519                         ibmf_status = ibmf_saa_check_sa_and_retry(
1520                             saa_portp, msgp, ibmf_callback, ibmf_callback_arg,
1521                             trans_info->si_trans_send_time, transport_flags);
1522                 }
```



```
1524                 if (ibmf_status != IBMF_SUCCESS)
1525                         break;

1527                 if (retry_count >= IBMF_SAA_MAX_BUSY_RETRY_COUNT)
1528                         break;

1530                 /* sync transaction with status SUCCESS should have response */
1531                 ASSERT(msgp->im_msgbufs_recv.im_bufs_mad_hdr != NULL);

1533                 mad_status = b2h16(msgp->im_msgbufs_recv.
1534                     im_bufs_mad_hdr->Status);

1536                 if ((mad_status != MAD_STATUS_BUSY) &&
1537                     (mad_status != MAD_STATUS_REDIRECT_REQUIRED))
1538                         break;

1540                 if (mad_status == MAD_STATUS_REDIRECT_REQUIRED) {

1542                         IBMF_TRACE_2(IBMF_TNF_DEBUG, DPRINT_L2,
1543                             ibmf_saa_impl_send_request, IBMF_TNF_TRACE, "",
1544                             "ibmf_saa_impl_send_request: %s, retry_count %d\n",
1545                             tnf_string, msg,
1546                             "response returned redirect status",
1547                             tnf_int, retry_count, retry_count);

1549                         /* update address info and copy it into msgp */
1550                         ibmf_saa_impl_update_sa_address_info(saa_portp, msgp);
1551                 } else {
1552                         IBMF_TRACE_2(IBMF_TNF_DEBUG, DPRINT_L2,
1553                             ibmf_saa_impl_send_request, IBMF_TNF_TRACE, "",
1554                             "ibmf_saa_impl_send_request: %s, retry_count %d\n",
1555                             tnf_string, msg, "response returned busy status",
1556                             tnf_int, retry_count, retry_count);
1557                 }

1559                 retry_count++;

1561                 /*
1562                  * since this is a blocking call, sleep for some time
1563                  * to allow SA to transition from busy state (if busy)
1564                  */
1565                 if (mad_status == MAD_STATUS_BUSY)
1566                         delay(drv_sectohz(IBMF_SAA_BUSY_RETRY_SLEEP_SECS));
1566                         delay(drv_usectohz(
1567                             IBMF_SAA_BUSY_RETRY_SLEEP_SECS * 1000000));
1567         }

1569         if (ibmf_status != IBMF_SUCCESS) {

1571                 IBMF_TRACE_2(IBMF_TNF_DEBUG, DPRINT_L2,
1572                     ibmf_saa_impl_send_request, IBMF_TNF_TRACE, "",
1573                     "ibmf_saa_impl_send_request: %s, ibmf_status = %d\n",
1574                     tnf_string, msg, "ibmf_msg_transport() failed",
1575                     tnf_int, ibmf_status, ibmf_status);

1577                 ibmf_saa_impl_free_msg(saa_portp->saa_pt_ibmf_handle, msgp);

1579                 mutex_enter(&saa_portp->saa_pt_mutex);

1581                 ASSERT(saa_portp->saa_pt_num_outstanding_trans > 0);
1582                 saa_portp->saa_pt_num_outstanding_trans--;

1584                 mutex_exit(&saa_portp->saa_pt_mutex);

1586                 if (client_data != NULL) {
```

```
1588                        mutex_enter(&client_data->saa_client_mutex);

1590                        ASSERT(client_data->saa_client_num_pending_trans > 0);
1591                        client_data->saa_client_num_pending_trans--;

1593                        if ((client_data->saa_client_num_pending_trans == 0) &&
1594                            (client_data->saa_client_state ==
1595                            SAA_CLIENT_STATE_WAITING))
1596                                cv_signal(&client_data->saa_client_state_cv);

1598                        mutex_exit(&client_data->saa_client_mutex);
1599                }

1601        } else if (sleep_flag == B_TRUE) {

1603                mutex_enter(&saa_portp->saa_pt_mutex);

1605                ASSERT(saa_portp->saa_pt_num_outstanding_trans > 0);
1606                saa_portp->saa_pt_num_outstanding_trans--;

1608                mutex_exit(&saa_portp->saa_pt_mutex);

1610                IBMF_TRACE_1(IBMF_TNF_DEBUG, DPRINT_L3,
1611                    ibmf_saa_impl_send_request, IBMF_TNF_TRACE, "",
1612                    "ibmf_saa_impl_send_request: %s\n",
1613                    tnf_string, msg, "Message sent and received successfully");

1615                /* fill in response values and free the message */
1616                ibmf_saa_impl_prepare_response(saa_portp->saa_pt_ibmf_handle,
1617                    msgp, B_FALSE, &trans_info->si_trans_status,
1618                    &trans_info->si_trans_result,
1619                    &trans_info->si_trans_length, sleep_flag);

1621                if (client_data != NULL) {
1622                        mutex_enter(&client_data->saa_client_mutex);

1624                        ASSERT(client_data->saa_client_num_pending_trans > 0);
1625                        client_data->saa_client_num_pending_trans--;

1627                        if ((client_data->saa_client_num_pending_trans == 0) &&
1628                            (client_data->saa_client_state ==
1629                            SAA_CLIENT_STATE_WAITING))
1630                                cv_signal(&client_data->saa_client_state_cv);

1632                        mutex_exit(&client_data->saa_client_mutex);
1633                }
1634        } else {

1636                IBMF_TRACE_1(IBMF_TNF_DEBUG, DPRINT_L3,
1637                    ibmf_saa_impl_send_request, IBMF_TNF_TRACE, "",
1638                    "ibmf_saa_impl_send_request: %s\n",
1639                    tnf_string, msg, "Message sent successfully");
1640        }

1642 bail:
1643        IBMF_TRACE_1(IBMF_TNF_DEBUG, DPRINT_L3,
1644            ibmf_saa_impl_send_request_end,
1645            IBMF_TNF_TRACE, "", "ibmf_saa_impl_send_request() exiting"
1646            " ibmf_status = %d\n", tnf_int, result, ibmf_status);

1648        return (ibmf_status);
1649 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  130313 Wed Aug 19 07:25:07 2015
new/usr/src/uts/common/io/igb/igb_main.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3438 #pragma inline(igb_arm_watchdog_timer)

3440 static void
3441 igb_arm_watchdog_timer(igb_t *igb)
3442 {
3443         /*
3444          * Fire a watchdog timer
3445          */
3446         igb->watchdog_tid =
3447             timeout(igb_local_timer,
3448             (void *)igb, drv_sectohz(1));
3448             (void *)igb, 1 * drv_usectohz(1000000));

3450 }
______unchanged_portion_omitted_
```

```
**********************************************************
   72139 Wed Aug 19 07:25:07 2015
new/usr/src/uts/common/io/ipw/ipw2100.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1008 static int
1009 ipw2100_cmd(struct ipw2100_softc *sc, uint32_t type, void *buf, size_t len)
1010 {
1011         struct ipw2100_bd       *txbd;
1012         clock_t                 clk;
1013         uint32_t                idx;

1015         /*
1016          * prepare command buffer
1017          */
1018         sc->sc_cmd->type = LE_32(type);
1019         sc->sc_cmd->subtype = LE_32(0);
1020         sc->sc_cmd->seq = LE_32(0);
1021         /*
1022          * copy data if any
1023          */
1024         if (len && buf)
1025                 (void) memcpy(sc->sc_cmd->data, buf, len);
1026         sc->sc_cmd->len = LE_32(len);

1028         /*
1029          * get host & device descriptor to submit command
1030          */
1031         mutex_enter(&sc->sc_tx_lock);

1033         IPW2100_DBG(IPW2100_DBG_RING, (sc->sc_dip, CE_CONT,
1034             "ipw2100_cmd(): tx-free=%d\n", sc->sc_tx_free));

1036         /*
1037          * command need 1 descriptor
1038          */
1039         while (sc->sc_tx_free < 1)  {
1040                 sc->sc_flags |= IPW2100_FLAG_CMD_WAIT;
1041                 cv_wait(&sc->sc_tx_cond, &sc->sc_tx_lock);
1042         }
1043         idx = sc->sc_tx_cur;

1045         IPW2100_DBG(IPW2100_DBG_RING, (sc->sc_dip, CE_CONT,
1046             "ipw2100_cmd(): tx-cur=%d\n", idx));

1048         sc->sc_done = 0;

1050         txbd            = &sc->sc_txbd[idx];
1051         txbd->phyaddr   = LE_32(sc->sc_dma_cmd.dr_pbase);
1052         txbd->len       = LE_32(sizeof (struct ipw2100_cmd));
1053         txbd->flags     = IPW2100_BD_FLAG_TX_FRAME_COMMAND
1054             | IPW2100_BD_FLAG_TX_LAST_FRAGMENT;
1055         txbd->nfrag     = 1;
1056         /*
1057          * sync for device
1058          */
1059         (void) ddi_dma_sync(sc->sc_dma_cmd.dr_hnd, 0,
1060             sizeof (struct ipw2100_cmd), DDI_DMA_SYNC_FORDEV);
1061         (void) ddi_dma_sync(sc->sc_dma_txbd.dr_hnd,
1062             idx * sizeof (struct ipw2100_bd),
1063             sizeof (struct ipw2100_bd), DDI_DMA_SYNC_FORDEV);

1065         /*
1066          * ring move forward
```



```
1067          */
1068         sc->sc_tx_cur = RING_FORWARD(sc->sc_tx_cur, 1, IPW2100_NUM_TXBD);
1069         sc->sc_tx_free--;
1070         ipw2100_csr_put32(sc, IPW2100_CSR_TX_WRITE_INDEX, sc->sc_tx_cur);
1071         mutex_exit(&sc->sc_tx_lock);

1073         /*
1074          * wait for command done
1075          */
1076         clk = drv_sectohz(1);
1076         clk = drv_usectohz(1000000);    /* 1 second */
1077         mutex_enter(&sc->sc_ilock);
1078         while (sc->sc_done == 0) {
1079                 /*
1080                  * pending for the response
1081                  */
1082                 if (cv_reltimedwait(&sc->sc_cmd_cond, &sc->sc_ilock,
1083                     clk, TR_CLOCK_TICK) < 0)
1084                         break;
1085         }
1086         mutex_exit(&sc->sc_ilock);

1088         IPW2100_DBG(IPW2100_DBG_RING, (sc->sc_dip, CE_CONT,
1089             "ipw2100_cmd(): cmd-done=%s\n", sc->sc_done ? "yes" : "no"));

1091         if (sc->sc_done == 0)
1092                 return (DDI_FAILURE);

1094         return (DDI_SUCCESS);
1095 }
______unchanged_portion_omitted_
```

```
**********************************************************
  153596 Wed Aug 19 07:25:07 2015
new/usr/src/uts/common/io/iwh/iwh.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


2140 /*
2141  * steps of loading ucode:
2142  * load init ucode=>init alive=>calibrate=>
2143  * receive calibration result=>reinitialize NIC=>
2144  * load runtime ucode=>runtime alive=>
2145  * send calibration result=>running.
2146  */
2147 static int
2148 iwh_load_init_firmware(iwh_sc_t *sc)
2149 {
2150         int err = IWH_FAIL;
2151         clock_t clk;

2153         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2155         /*
2156          * load init_text section of uCode to hardware
2157          */
2158         err = iwh_put_seg_fw(sc, sc->sc_dma_fw_init_text.cookie.dmac_address,
2159             RTC_INST_LOWER_BOUND, sc->sc_dma_fw_init_text.cookie.dmac_size);
2160         if (err != IWH_SUCCESS) {
2161                 cmn_err(CE_WARN, "iwh_load_init_firmware(): "
2162                     "failed to write init uCode.\n");
2163                 return (err);
2164         }

2166         clk = ddi_get_lbolt() + drv_sectohz(1);
2166         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2168         /*
2169          * wait loading init_text until completed or timeout
2170          */
2171         while (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2172                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2173                         break;
2174                 }
2175         }

2177         if (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2178                 cmn_err(CE_WARN, "iwh_load_init_firmware(): "
2179                     "timeout waiting for init uCode load.\n");
2180                 return (IWH_FAIL);
2181         }

2183         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2185         /*
2186          * load init_data section of uCode to hardware
2187          */
2188         err = iwh_put_seg_fw(sc, sc->sc_dma_fw_init_data.cookie.dmac_address,
2189             RTC_DATA_LOWER_BOUND, sc->sc_dma_fw_init_data.cookie.dmac_size);
2190         if (err != IWH_SUCCESS) {
2191                 cmn_err(CE_WARN, "iwh_load_init_firmware(): "
2192                     "failed to write init_data uCode.\n");
2193                 return (err);
2194         }

2196         clk = ddi_get_lbolt() + drv_sectohz(1);
```



```
2196         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2198         /*
2199          * wait loading init_data until completed or timeout
2200          */
2201         while (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2202                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2203                         break;
2204                 }
2205         }

2207         if (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2208                 cmn_err(CE_WARN, "iwh_load_init_firmware(): "
2209                     "timeout waiting for init_data uCode load.\n");
2210                 return (IWH_FAIL);
2211         }

2213         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2215         return (err);
2216 }

2218 static int
2219 iwh_load_run_firmware(iwh_sc_t *sc)
2220 {
2221         int err = IWH_FAIL;
2222         clock_t clk;

2224         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2226         /*
2227          * load init_text section of uCode to hardware
2228          */
2229         err = iwh_put_seg_fw(sc, sc->sc_dma_fw_text.cookie.dmac_address,
2230             RTC_INST_LOWER_BOUND, sc->sc_dma_fw_text.cookie.dmac_size);
2231         if (err != IWH_SUCCESS) {
2232                 cmn_err(CE_WARN, "iwh_load_run_firmware(): "
2233                     "failed to write run uCode.\n");
2234                 return (err);
2235         }

2237         clk = ddi_get_lbolt() + drv_sectohz(1);
2237         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2239         /*
2240          * wait loading run_text until completed or timeout
2241          */
2242         while (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2243                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2244                         break;
2245                 }
2246         }

2248         if (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2249                 cmn_err(CE_WARN, "iwh_load_run_firmware(): "
2250                     "timeout waiting for run uCode load.\n");
2251                 return (IWH_FAIL);
2252         }

2254         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2256         /*
2257          * load run_data section of uCode to hardware
2258          */
2259         err = iwh_put_seg_fw(sc, sc->sc_dma_fw_data_bak.cookie.dmac_address,
2260             RTC_DATA_LOWER_BOUND, sc->sc_dma_fw_data.cookie.dmac_size);
```

```
2261         if (err != IWH_SUCCESS) {
2262                 cmn_err(CE_WARN, "iwh_load_run_firmware(): "
2263                     "failed to write run_data uCode.\n");
2264                 return (err);
2265         }

2267         clk = ddi_get_lbolt() + drv_sectohz(1);
2267         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2269         /*
2270          * wait loading run_data until completed or timeout
2271          */
2272         while (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2273                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2274                         break;
2275                 }
2276         }

2278         if (!(sc->sc_flags & IWH_F_PUT_SEG)) {
2279                 cmn_err(CE_WARN, "iwh_load_run_firmware(): "
2280                     "timeout waiting for run_data uCode load.\n");
2281                 return (IWH_FAIL);
2282         }

2284         atomic_and_32(&sc->sc_flags, ~IWH_F_PUT_SEG);

2286         return (err);
2287 }
_____unchanged_portion_omitted_

3780 /*
3781  * kernel thread to deal with exceptional situation
3782  */
3783 static void
3784 iwh_thread(iwh_sc_t *sc)
3785 {
3786         ieee80211com_t *ic = &sc->sc_ic;
3787         clock_t clk;
3788         int err, n = 0, timeout = 0;
3789         uint32_t tmp;
3790 #ifdef  DEBUG
3791         int times = 0;
3792 #endif

3794         while (sc->sc_mf_thread_switch) {
3795                 tmp = IWH_READ(sc, CSR_GP_CNTRL);
3796                 if (tmp & CSR_GP_CNTRL_REG_FLAG_HW_RF_KILL_SW) {
3797                         atomic_and_32(&sc->sc_flags, ~IWH_F_RADIO_OFF);
3798                 } else {
3799                         atomic_or_32(&sc->sc_flags, IWH_F_RADIO_OFF);
3800                 }

3802                 /*
3803                  * If  in SUSPEND or the RF is OFF, do nothing.
3804                  */
3805                 if (sc->sc_flags & IWH_F_RADIO_OFF) {
3806                         delay(drv_usectohz(100000));
3807                         continue;
3808                 }

3810                 /*
3811                  * recovery fatal error
3812                  */
3813                 if (ic->ic_mach &&
3814                     (sc->sc_flags & IWH_F_HW_ERR_RECOVER)) {
```



```
3816                         IWH_DBG((IWH_DEBUG_FW, "iwh_thread(): "
3817                             "try to recover fatal hw error: %d\n", times++));

3819                         iwh_stop(sc);

3821                         if (IWH_CHK_FAST_RECOVER(sc)) {
3822                                 /*
3823                                  * save runtime configuration
3824                                  */
3825                                 bcopy(&sc->sc_config, &sc->sc_config_save,
3826                                     sizeof (sc->sc_config));
3827                         } else {
3828                                 ieee80211_new_state(ic, IEEE80211_S_INIT, -1);
3829                                 delay(drv_usectohz(2000000 + n*500000));
3830                         }

3832                         err = iwh_init(sc);
3833                         if (err != IWH_SUCCESS) {
3834                                 n++;
3835                                 if (n < 20) {
3836                                         continue;
3837                                 }
3838                         }

3840                         n = 0;
3841                         if (!err) {
3842                                 atomic_or_32(&sc->sc_flags, IWH_F_RUNNING);
3843                         }


3846                         if (!IWH_CHK_FAST_RECOVER(sc) ||
3847                             iwh_fast_recover(sc) != IWH_SUCCESS) {
3848                                 atomic_and_32(&sc->sc_flags,
3849                                     ~IWH_F_HW_ERR_RECOVER);

3851                                 delay(drv_usectohz(2000000));
3852                                 if (sc->sc_ostate != IEEE80211_S_INIT) {
3853                                         ieee80211_new_state(ic,
3854                                             IEEE80211_S_SCAN, 0);
3855                                 }
3856                         }
3857                 }

3859                 if (ic->ic_mach &&
3860                     (sc->sc_flags & IWH_F_SCANNING) && sc->sc_scan_pending) {
3861                         IWH_DBG((IWH_DEBUG_SCAN, "iwh_thread(): "
3862                             "wait for probe response\n"));

3864                         sc->sc_scan_pending--;
3865                         delay(drv_usectohz(200000));
3866                         ieee80211_next_scan(ic);
3867                 }

3869                 /*
3870                  * rate ctl
3871                  */
3872                 if (ic->ic_mach &&
3873                     (sc->sc_flags & IWH_F_RATE_AUTO_CTL)) {
3874                         clk = ddi_get_lbolt();
3875                         if (clk > sc->sc_clk + drv_sectohz(1)) {
3875                         if (clk > sc->sc_clk + drv_usectohz(1000000)) {
3876                                 iwh_amrr_timeout(sc);
3877                         }
3878                 }

3880                 if ((ic->ic_state == IEEE80211_S_RUN) &&
```

```
3881                    (ic->ic_beaconmiss++ > 100)) {      /* 10 seconds */
3882                        cmn_err(CE_WARN, "iwh: beacon missed for 10 seconds\n");
3883                        (void) ieee80211_new_state(ic,
3884                            IEEE80211_S_INIT, -1);
3885                }

3887                delay(drv_usectohz(100000));

3889                mutex_enter(&sc->sc_mt_lock);
3890                if (sc->sc_tx_timer) {
3891                        timeout++;
3892                        if (10 == timeout) {
3893                                sc->sc_tx_timer--;
3894                                if (0 == sc->sc_tx_timer) {
3895                                        atomic_or_32(&sc->sc_flags,
3896                                            IWH_F_HW_ERR_RECOVER);
3897                                        sc->sc_ostate = IEEE80211_S_RUN;
3898                                        IWH_DBG((IWH_DEBUG_FW, "iwh_thread(): "
3899                                            "try to recover from "
3900                                            "send fail\n"));
3901                                }
3902                                timeout = 0;
3903                        }
3904                }
3905                mutex_exit(&sc->sc_mt_lock);
3906        }

3908        mutex_enter(&sc->sc_mt_lock);
3909        sc->sc_mf_thread = NULL;
3910        cv_signal(&sc->sc_mt_cv);
3911        mutex_exit(&sc->sc_mt_lock);
3912 }
```

unchanged_portion_omitted_

```
4788 /*
4789  * main initialization function
4790  */
4791 static int
4792 iwh_init(iwh_sc_t *sc)
4793 {
4794        int err = IWH_FAIL;
4795        clock_t clk;

4797        /*
4798         * release buffer for calibration
4799         */
4800        iwh_release_calib_buffer(sc);

4802        mutex_enter(&sc->sc_glock);
4803        atomic_and_32(&sc->sc_flags, ~IWH_F_FW_INIT);

4805        err = iwh_init_common(sc);
4806        if (err != IWH_SUCCESS) {
4807                mutex_exit(&sc->sc_glock);
4808                return (IWH_FAIL);
4809        }

4811        /*
4812         * backup ucode data part for future use.
4813         */
4814        bcopy(sc->sc_dma_fw_data.mem_va,
4815            sc->sc_dma_fw_data_bak.mem_va,
4816            sc->sc_dma_fw_data.alength);

4818        /* load firmware init segment into NIC */
4819        err = iwh_load_init_firmware(sc);
```



```
4820        if (err != IWH_SUCCESS) {
4821                cmn_err(CE_WARN, "iwh_init(): "
4822                    "failed to setup init firmware\n");
4823                mutex_exit(&sc->sc_glock);
4824                return (IWH_FAIL);
4825        }

4827        /*
4828         * now press "execute" start running
4829         */
4830        IWH_WRITE(sc, CSR_RESET, 0);

4832        clk = ddi_get_lbolt() + drv_sectohz(1);
4832        clk = ddi_get_lbolt() + drv_usectohz(1000000);
4833        while (!(sc->sc_flags & IWH_F_FW_INIT)) {
4834                if (cv_timedwait(&sc->sc_ucode_cv,
4835                    &sc->sc_glock, clk) < 0) {
4836                        break;
4837                }
4838        }

4840        if (!(sc->sc_flags & IWH_F_FW_INIT)) {
4841                cmn_err(CE_WARN, "iwh_init(): "
4842                    "failed to process init alive.\n");
4843                mutex_exit(&sc->sc_glock);
4844                return (IWH_FAIL);
4845        }

4847        mutex_exit(&sc->sc_glock);

4849        /*
4850         * stop chipset for initializing chipset again
4851         */
4852        iwh_stop(sc);

4854        mutex_enter(&sc->sc_glock);
4855        atomic_and_32(&sc->sc_flags, ~IWH_F_FW_INIT);

4857        err = iwh_init_common(sc);
4858        if (err != IWH_SUCCESS) {
4859                mutex_exit(&sc->sc_glock);
4860                return (IWH_FAIL);
4861        }

4863        /*
4864         * load firmware run segment into NIC
4865         */
4866        err = iwh_load_run_firmware(sc);
4867        if (err != IWH_SUCCESS) {
4868                cmn_err(CE_WARN, "iwh_init(): "
4869                    "failed to setup run firmware\n");
4870                mutex_exit(&sc->sc_glock);
4871                return (IWH_FAIL);
4872        }

4874        /*
4875         * now press "execute" start running
4876         */
4877        IWH_WRITE(sc, CSR_RESET, 0);

4879        clk = ddi_get_lbolt() + drv_sectohz(1);
4879        clk = ddi_get_lbolt() + drv_usectohz(1000000);
4880        while (!(sc->sc_flags & IWH_F_FW_INIT)) {
4881                if (cv_timedwait(&sc->sc_ucode_cv,
4882                    &sc->sc_glock, clk) < 0) {
4883                        break;
```

```
4884                 }
4885         }

4887         if (!(sc->sc_flags & IWH_F_FW_INIT)) {
4888                 cmn_err(CE_WARN, "iwh_init(): "
4889                     "failed to process runtime alive.\n");
4890                 mutex_exit(&sc->sc_glock);
4891                 return (IWH_FAIL);
4892         }

4894         mutex_exit(&sc->sc_glock);

4896         DELAY(1000);

4898         mutex_enter(&sc->sc_glock);
4899         atomic_and_32(&sc->sc_flags, ~IWH_F_FW_INIT);

4901         /*
4902          * at this point, the firmware is loaded OK, then config the hardware
4903          * with the ucode API, including rxon, txpower, etc.
4904          */
4905         err = iwh_config(sc);
4906         if (err) {
4907                 cmn_err(CE_WARN, "iwh_init(): "
4908                     "failed to configure device\n");
4909                 mutex_exit(&sc->sc_glock);
4910                 return (IWH_FAIL);
4911         }

4913         /*
4914          * at this point, hardware may receive beacons :)
4915          */
4916         mutex_exit(&sc->sc_glock);
4917         return (IWH_SUCCESS);
4918 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  126747 Wed Aug 19 07:25:08 2015
new/usr/src/uts/common/io/iwp/iwp.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


2047 /*
2048  * steps of loading ucode:
2049  * load init ucode=>init alive=>calibrate=>
2050  * receive calibration result=>reinitialize NIC=>
2051  * load runtime ucode=>runtime alive=>
2052  * send calibration result=>running.
2053  */
2054 static int
2055 iwp_load_init_firmware(iwp_sc_t *sc)
2056 {
2057         int     err = IWP_FAIL;
2058         clock_t clk;

2060         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2062         /*
2063          * load init_text section of uCode to hardware
2064          */
2065         err = iwp_put_seg_fw(sc, sc->sc_dma_fw_init_text.cookie.dmac_address,
2066             RTC_INST_LOWER_BOUND, sc->sc_dma_fw_init_text.cookie.dmac_size);
2067         if (err != IWP_SUCCESS) {
2068                 cmn_err(CE_WARN, "iwp_load_init_firmware(): "
2069                     "failed to write init uCode.\n");
2070                 return (err);
2071         }

2073         clk = ddi_get_lbolt() + drv_sectohz(1);
2073         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2075         /* wait loading init_text until completed or timeout */
2076         while (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2077                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2078                         break;
2079                 }
2080         }

2082         if (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2083                 cmn_err(CE_WARN, "iwp_load_init_firmware(): "
2084                     "timeout waiting for init uCode load.\n");
2085                 return (IWP_FAIL);
2086         }

2088         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2090         /*
2091          * load init_data section of uCode to hardware
2092          */
2093         err = iwp_put_seg_fw(sc, sc->sc_dma_fw_init_data.cookie.dmac_address,
2094             RTC_DATA_LOWER_BOUND, sc->sc_dma_fw_init_data.cookie.dmac_size);
2095         if (err != IWP_SUCCESS) {
2096                 cmn_err(CE_WARN, "iwp_load_init_firmware(): "
2097                     "failed to write init_data uCode.\n");
2098                 return (err);
2099         }

2101         clk = ddi_get_lbolt() + drv_sectohz(1);
2101         clk = ddi_get_lbolt() + drv_usectohz(1000000);
```



```
2103         /*
2104          * wait loading init_data until completed or timeout
2105          */
2106         while (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2107                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2108                         break;
2109                 }
2110         }

2112         if (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2113                 cmn_err(CE_WARN, "iwp_load_init_firmware(): "
2114                     "timeout waiting for init_data uCode load.\n");
2115                 return (IWP_FAIL);
2116         }

2118         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2120         return (err);
2121 }

2123 static int
2124 iwp_load_run_firmware(iwp_sc_t *sc)
2125 {
2126         int     err = IWP_FAIL;
2127         clock_t clk;

2129         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2131         /*
2132          * load init_text section of uCode to hardware
2133          */
2134         err = iwp_put_seg_fw(sc, sc->sc_dma_fw_text.cookie.dmac_address,
2135             RTC_INST_LOWER_BOUND, sc->sc_dma_fw_text.cookie.dmac_size);
2136         if (err != IWP_SUCCESS) {
2137                 cmn_err(CE_WARN, "iwp_load_run_firmware(): "
2138                     "failed to write run uCode.\n");
2139                 return (err);
2140         }

2142         clk = ddi_get_lbolt() + drv_sectohz(1);
2142         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2144         /* wait loading run_text until completed or timeout */
2145         while (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2146                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2147                         break;
2148                 }
2149         }

2151         if (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2152                 cmn_err(CE_WARN, "iwp_load_run_firmware(): "
2153                     "timeout waiting for run uCode load.\n");
2154                 return (IWP_FAIL);
2155         }

2157         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2159         /*
2160          * load run_data section of uCode to hardware
2161          */
2162         err = iwp_put_seg_fw(sc, sc->sc_dma_fw_data_bak.cookie.dmac_address,
2163             RTC_DATA_LOWER_BOUND, sc->sc_dma_fw_data.cookie.dmac_size);
2164         if (err != IWP_SUCCESS) {
2165                 cmn_err(CE_WARN, "iwp_load_run_firmware(): "
2166                     "failed to write run_data uCode.\n");
2167                 return (err);
```

```
2168         }

2170         clk = ddi_get_lbolt() + drv_sectohz(1);
2170         clk = ddi_get_lbolt() + drv_usectohz(1000000);

2172         /*
2173          * wait loading run_data until completed or timeout
2174          */
2175         while (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2176                 if (cv_timedwait(&sc->sc_put_seg_cv, &sc->sc_glock, clk) < 0) {
2177                         break;
2178                 }
2179         }

2181         if (!(sc->sc_flags & IWP_F_PUT_SEG)) {
2182                 cmn_err(CE_WARN, "iwp_load_run_firmware(): "
2183                     "timeout waiting for run_data uCode load.\n");
2184                 return (IWP_FAIL);
2185         }

2187         atomic_and_32(&sc->sc_flags, ~IWP_F_PUT_SEG);

2189         return (err);
2190 }
```

*unchanged_portion_omitted_*

```
3607 /*
3608  * kernel thread to deal with exceptional situation
3609  */
3610 static void
3611 iwp_thread(iwp_sc_t *sc)
3612 {
3613         ieee80211com_t  *ic = &sc->sc_ic;
3614         clock_t clk;
3615         int err, n = 0, timeout = 0;
3616         uint32_t tmp;
3617 #ifdef  DEBUG
3618         int times = 0;
3619 #endif

3621         while (sc->sc_mf_thread_switch) {
3622                 tmp = IWP_READ(sc, CSR_GP_CNTRL);
3623                 if (tmp & CSR_GP_CNTRL_REG_FLAG_HW_RF_KILL_SW) {
3624                         atomic_and_32(&sc->sc_flags, ~IWP_F_RADIO_OFF);
3625                 } else {
3626                         atomic_or_32(&sc->sc_flags, IWP_F_RADIO_OFF);
3627                 }

3629                 /*
3630                  * If  in SUSPEND or the RF is OFF, do nothing.
3631                  */
3632                 if (sc->sc_flags & IWP_F_RADIO_OFF) {
3633                         delay(drv_usectohz(100000));
3634                         continue;
3635                 }

3637                 /*
3638                  * recovery fatal error
3639                  */
3640                 if (ic->ic_mach &&
3641                     (sc->sc_flags & IWP_F_HW_ERR_RECOVER)) {

3643                         IWP_DBG((IWP_DEBUG_FW, "iwp_thread(): "
3644                             "try to recover fatal hw error: %d\n", times++));

3646                         iwp_stop(sc);
```



```
3648                         if (IWP_CHK_FAST_RECOVER(sc)) {
3649                                 /* save runtime configuration */
3650                                 bcopy(&sc->sc_config, &sc->sc_config_save,
3651                                     sizeof (sc->sc_config));
3652                         } else {
3653                                 ieee80211_new_state(ic, IEEE80211_S_INIT, -1);
3654                                 delay(drv_usectohz(2000000 + n*500000));
3655                         }

3657                         err = iwp_init(sc);
3658                         if (err != IWP_SUCCESS) {
3659                                 n++;
3660                                 if (n < 20) {
3661                                         continue;
3662                                 }
3663                         }

3665                         n = 0;
3666                         if (!err) {
3667                                 atomic_or_32(&sc->sc_flags, IWP_F_RUNNING);
3668                         }


3671                         if (!IWP_CHK_FAST_RECOVER(sc) ||
3672                             iwp_fast_recover(sc) != IWP_SUCCESS) {
3673                                 atomic_and_32(&sc->sc_flags,
3674                                     ~IWP_F_HW_ERR_RECOVER);

3676                                 delay(drv_usectohz(2000000));
3677                                 if (sc->sc_ostate != IEEE80211_S_INIT) {
3678                                         ieee80211_new_state(ic,
3679                                             IEEE80211_S_SCAN, 0);
3680                                 }
3681                         }
3682                 }

3684                 if (ic->ic_mach &&
3685                     (sc->sc_flags & IWP_F_SCANNING) && sc->sc_scan_pending) {
3686                         IWP_DBG((IWP_DEBUG_SCAN, "iwp_thread(): "
3687                             "wait for probe response\n"));

3689                         sc->sc_scan_pending--;
3690                         delay(drv_usectohz(200000));
3691                         ieee80211_next_scan(ic);
3692                 }

3694                 /*
3695                  * rate ctl
3696                  */
3697                 if (ic->ic_mach &&
3698                     (sc->sc_flags & IWP_F_RATE_AUTO_CTL)) {
3699                         clk = ddi_get_lbolt();
3700                         if (clk > sc->sc_clk + drv_sectohz(1)) {
3700                         if (clk > sc->sc_clk + drv_usectohz(1000000)) {
3701                                 iwp_amrr_timeout(sc);
3702                         }
3703                 }

3705                 delay(drv_usectohz(100000));

3707                 mutex_enter(&sc->sc_mt_lock);
3708                 if (sc->sc_tx_timer) {
3709                         timeout++;
3710                         if (10 == timeout) {
3711                                 sc->sc_tx_timer--;
```

```
3712                                if (0 == sc->sc_tx_timer) {
3713                                        atomic_or_32(&sc->sc_flags,
3714                                            IWP_F_HW_ERR_RECOVER);
3715                                        sc->sc_ostate = IEEE80211_S_RUN;
3716                                        IWP_DBG((IWP_DEBUG_FW, "iwp_thread(): "
3717                                            "try to recover from "
3718                                            "send fail\n"));
3719                                }
3720                                timeout = 0;
3721                        }
3722                }
3723                mutex_exit(&sc->sc_mt_lock);
3724        }

3726        mutex_enter(&sc->sc_mt_lock);
3727        sc->sc_mf_thread = NULL;
3728        cv_signal(&sc->sc_mt_cv);
3729        mutex_exit(&sc->sc_mt_lock);
3730 }
_____unchanged_portion_omitted_

4524 /*
4525  * main initialization function
4526  */
4527 static int
4528 iwp_init(iwp_sc_t *sc)
4529 {
4530        int err = IWP_FAIL;
4531        clock_t clk;

4533        /*
4534         * release buffer for calibration
4535         */
4536        iwp_release_calib_buffer(sc);

4538        mutex_enter(&sc->sc_glock);
4539        atomic_and_32(&sc->sc_flags, ~IWP_F_FW_INIT);

4541        err = iwp_init_common(sc);
4542        if (err != IWP_SUCCESS) {
4543                mutex_exit(&sc->sc_glock);
4544                return (IWP_FAIL);
4545        }

4547        /*
4548         * backup ucode data part for future use.
4549         */
4550        (void) memcpy(sc->sc_dma_fw_data_bak.mem_va,
4551            sc->sc_dma_fw_data.mem_va,
4552            sc->sc_dma_fw_data.alength);

4554        /* load firmware init segment into NIC */
4555        err = iwp_load_init_firmware(sc);
4556        if (err != IWP_SUCCESS) {
4557                cmn_err(CE_WARN, "iwp_init(): "
4558                    "failed to setup init firmware\n");
4559                mutex_exit(&sc->sc_glock);
4560                return (IWP_FAIL);
4561        }

4563        /*
4564         * now press "execute" start running
4565         */
4566        IWP_WRITE(sc, CSR_RESET, 0);

4568        clk = ddi_get_lbolt() + drv_sectohz(1);
```



```
4568        clk = ddi_get_lbolt() + drv_usectohz(1000000);
4569        while (!(sc->sc_flags & IWP_F_FW_INIT)) {
4570                if (cv_timedwait(&sc->sc_ucode_cv,
4571                    &sc->sc_glock, clk) < 0) {
4572                        break;
4573                }
4574        }

4576        if (!(sc->sc_flags & IWP_F_FW_INIT)) {
4577                cmn_err(CE_WARN, "iwp_init(): "
4578                    "failed to process init alive.\n");
4579                mutex_exit(&sc->sc_glock);
4580                return (IWP_FAIL);
4581        }

4583        mutex_exit(&sc->sc_glock);

4585        /*
4586         * stop chipset for initializing chipset again
4587         */
4588        iwp_stop(sc);

4590        mutex_enter(&sc->sc_glock);
4591        atomic_and_32(&sc->sc_flags, ~IWP_F_FW_INIT);

4593        err = iwp_init_common(sc);
4594        if (err != IWP_SUCCESS) {
4595                mutex_exit(&sc->sc_glock);
4596                return (IWP_FAIL);
4597        }

4599        /*
4600         * load firmware run segment into NIC
4601         */
4602        err = iwp_load_run_firmware(sc);
4603        if (err != IWP_SUCCESS) {
4604                cmn_err(CE_WARN, "iwp_init(): "
4605                    "failed to setup run firmware\n");
4606                mutex_exit(&sc->sc_glock);
4607                return (IWP_FAIL);
4608        }

4610        /*
4611         * now press "execute" start running
4612         */
4613        IWP_WRITE(sc, CSR_RESET, 0);

4615        clk = ddi_get_lbolt() + drv_sectohz(1);
4615        clk = ddi_get_lbolt() + drv_usectohz(1000000);
4616        while (!(sc->sc_flags & IWP_F_FW_INIT)) {
4617                if (cv_timedwait(&sc->sc_ucode_cv,
4618                    &sc->sc_glock, clk) < 0) {
4619                        break;
4620                }
4621        }

4623        if (!(sc->sc_flags & IWP_F_FW_INIT)) {
4624                cmn_err(CE_WARN, "iwp_init(): "
4625                    "failed to process runtime alive.\n");
4626                mutex_exit(&sc->sc_glock);
4627                return (IWP_FAIL);
4628        }

4630        mutex_exit(&sc->sc_glock);

4632        DELAY(1000);
```

```
4634         mutex_enter(&sc->sc_glock);
4635         atomic_and_32(&sc->sc_flags, ~IWP_F_FW_INIT);

4637         /*
4638          * at this point, the firmware is loaded OK, then config the hardware
4639          * with the ucode API, including rxon, txpower, etc.
4640          */
4641         err = iwp_config(sc);
4642         if (err) {
4643                 cmn_err(CE_WARN, "iwp_init(): "
4644                     "failed to configure device\n");
4645                 mutex_exit(&sc->sc_glock);
4646                 return (IWP_FAIL);
4647         }

4649         /*
4650          * at this point, hardware may receive beacons :)
4651          */
4652         mutex_exit(&sc->sc_glock);
4653         return (IWP_SUCCESS);
4654 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  143652 Wed Aug 19 07:25:08 2015
new/usr/src/uts/common/io/ixgbe/ixgbe_main.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3699 #pragma inline(ixgbe_arm_watchdog_timer)
3700 static void
3701 ixgbe_arm_watchdog_timer(ixgbe_t *ixgbe)
3702 {
3703         /*
3704          * Fire a watchdog timer
3705          */
3706         ixgbe->watchdog_tid =
3707             timeout(ixgbe_local_timer,
3708             (void *)ixgbe, drv_sectohz(1));
3708             (void *)ixgbe, 1 * drv_usectohz(1000000));

3710 }
______unchanged_portion_omitted_
```

```
**********************************************************
   43834 Wed Aug 19 07:25:08 2015
new/usr/src/uts/common/io/lvm/md/md_rename.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1000 /*
1001  * The order of the called role swap functions is critical.
1002  * If they're not ordered as "all parents", then "all self"
1003  * then "all child" transitions, we will almost certainly
1004  * corrupt the data base and the in-core linkages. So,
1005  * verify that the list built by the individual drivers is
1006  * ok here.
1007  *
1008  * We could have done fancy bit encodings of the roles so
1009  * it all fit into a single word and we wouldn't need the
1010  * prev_ord field. But, since cpu power is cheaper than
1011  * than people power, they're all separate for easier
1012  * debugging and maintaining. (In the unlikely event that
1013  * rename/exchange ever becomes cpu-limited, and this
1014  * algorithm is the bottleneck, we should revisit this.)
1015  */

1017 static bool_t
1018 role_swap_is_valid(
1019         int              previous,
1020         int              current,
1021         md_rendelta_t   *delta,
1022         md_rentxn_t     *rtxnp)
1023 {
1024         bool_t  valid   = FALSE;

1026         /*
1027          * we've backed up in processing the role table
1028          */
1029         if ((previous > current) &&
1030             (delta->prev && (delta->old_role != delta->prev->old_role))) {
1031                 goto out;
1032         }

1034         /*
1035          * we're repeating the same role transition
1036          */
1037         if (previous == current) {
1038                 switch (delta->old_role) {
1039                 case MDRR_PARENT:
1040                         /*
1041                          * require at least one of the devices to
1042                          * be multiparented for us to allow another
1043                          * parent transition
1044                          */
1045                         if ((MD_MULTI_PARENT != MD_PARENT(rtxnp->from.unp)) &&
1046                             (MD_MULTI_PARENT != MD_PARENT(rtxnp->to.unp))) {
1047                                 goto out;
1048                         }
1049                         break;

1051                 case MDRR_CHILD:
1052                         /* it's ok to have multiple children */
1053                         break;

1055                 case MDRR_SELF:
1056                         /* it's never ok to have multiple self transitions */
1057                         /* FALLTHROUGH */
1058                 default:
```



```
1059                         goto out;
1060                 }
1061         }

1063         valid = TRUE;
1064 out:
1065         if (!valid) {
1066                 if (md_rename_debug != 0) {
1067                         cmn_err(CE_NOTE, "previous: %d, current: %d, role: %s",
1068                                 previous, current,
1069                                 ROLE(delta->old_role));
1070                         delay(drv_sectohz(3));
1070                         delay(3*drv_usectohz(1000000));
1071                         ASSERT(FALSE);
1072                 }
1073         }

1075         return (valid);
1076 }
______unchanged_portion_omitted_

1188 /*
1189  * dump contents of rename transaction
1190  */
1191 static void
1192 dump_txn(md_rentxn_t *rtxnp) {

1194         if (md_rename_debug == 0) {
1195                 return;
1196         }

1198         cmn_err(CE_NOTE, "rtxnp: %p", (void *) rtxnp);
1199         if (rtxnp) {
1200                 cmn_err(CE_NOTE, "beginning: %llx, op: %s",
1201                         rtxnp->beginning, OP_STR(rtxnp->op));

1203                 cmn_err(CE_NOTE,
1204         "revision: %d, uflags: %d, rec_idx: %d, n_recids: %d, rec_ids: %p%s",
1205                         rtxnp->revision, rtxnp->uflags,
1206                         rtxnp->rec_idx, rtxnp->n_recids, (void *) rtxnp->recids,
1207                         rtxnp->stat.trans_in_stack? " (trans in stack)": "");
1208                 cmn_err(CE_NOTE, " from: beginning: %llx",
1209                                                         rtxnp->from.beginning);
1210                 cmn_err(CE_NOTE, "    minor: %lX, key: %lX",
1211                         (ulong_t)rtxnp->from.mnum, (ulong_t)rtxnp->from.key);
1212                 cmn_err(CE_NOTE, "    unp: %lX, uip: %lX",
1213                         (ulong_t)rtxnp->from.unp, (ulong_t)rtxnp->from.uip);
1214                 cmn_err(CE_NOTE, "    end: %llx", rtxnp->from.end);
1215                 cmn_err(CE_NOTE, "  to: beginning: %llx", rtxnp->to.beginning);
1216                 cmn_err(CE_NOTE, "    minor: %lX, key: %lX",
1217                         (ulong_t)rtxnp->to.mnum, (ulong_t)rtxnp->to.key);
1218                 cmn_err(CE_NOTE, "    unp: %lX, uip: %lX",
1219                         (ulong_t)rtxnp->to.unp, (ulong_t)rtxnp->to.uip);
1220                 cmn_err(CE_NOTE, "    end: %llx", rtxnp->to.end);
1221                 cmn_err(CE_NOTE, "end: %llx\n", rtxnp->end);
1222         }
1223         delay(drv_sectohz(1));
1223         delay(drv_usectohz(1000000));
1224 }

1226 /*
1227  * dump contents of all deltas
1228  */
1229 static void
1230 dump_dtree(md_rendelta_t *family)
1231 {
```

```
1232         md_rendelta_t   *r;
1233         int             i;

1235         if (md_rename_debug == 0) {
1236                 return;
1237         }

1239         for (r = family, i = 0; r; r = r->next, i++) {
1240                 cmn_err(CE_NOTE, "%d.  beginning: %llx", i, r->beginning);
1241                 cmn_err(CE_NOTE, "  r: %lX, dev: %lX, next: %lx, prev: %lx",
1242                                     (ulong_t)r, (ulong_t)r->dev,
1243                                     (ulong_t)r->next, (ulong_t)r->prev);

1245                 cmn_err(CE_NOTE, "  role: %s -> %s, unp: %lx, uip: %lx",
1246                         ROLE(r->old_role), ROLE(r->new_role),
1247                         (ulong_t)r->unp, (ulong_t)r->uip);
1248                 cmn_err(CE_NOTE,
1249                 "  lock: %lx, unlock: %lx\n\t  check: %lx, role_swap: %lx",
1250                         (ulong_t)r->lock, (ulong_t)r->unlock,
1251                         (ulong_t)r->check, (ulong_t)r->role_swap);
1252                 if (*((uint_t *)(&r->txn_stat)) != 0) {
1253                         cmn_err(CE_NOTE, "status: (0x%x) %s%s%s%s%s",
1254                         *((uint_t *)(&r->txn_stat)),
1255                         r->txn_stat.is_open?            "is_open "      : "",
1256                         r->txn_stat.locked?             "locked "       : "",
1257                         r->txn_stat.checked?            "checked "      : "",
1258                         r->txn_stat.role_swapped?       "role_swapped " : "",
1259                         r->txn_stat.unlocked?           "unlocked"      : "");
1260                 }
1261                 cmn_err(CE_NOTE, "end: %llx\n", r->end);
1262         }
1263         delay(drv_sectohz(1));
1263         delay(drv_usectohz(1000000));
1264 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  106477 Wed Aug 19 07:25:08 2015
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 888 static int
 889 mrsas_tbolt_ioc_init(struct mrsas_instance *instance, dma_obj_t *mpi2_dma_obj)
 890 {
 891         int                             numbytes;
 892         uint16_t                        flags;
 893         struct mrsas_init_frame2        *mfiFrameInit2;
 894         struct mrsas_header             *frame_hdr;
 895         Mpi2IOCInitRequest_t            *init;
 896         struct mrsas_cmd                *cmd = NULL;
 897         struct mrsas_drv_ver            drv_ver_info;
 898         MRSAS_REQUEST_DESCRIPTOR_UNION  req_desc;
 899         uint32_t                        timeout;

 901         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


 904 #ifdef DEBUG
 905         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
 906             (int)sizeof (*mfiFrameInit2)));
 907         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", (int)sizeof (*init)));
 908         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
 909             (int)sizeof (struct mrsas_init_frame2)));
 910         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n",
 911             (int)sizeof (Mpi2IOCInitRequest_t)));
 912 #endif

 914         init = (Mpi2IOCInitRequest_t *)mpi2_dma_obj->buffer;
 915         numbytes = sizeof (*init);
 916         bzero(init, numbytes);

 918         ddi_put8(mpi2_dma_obj->acc_handle, &init->Function,
 919             MPI2_FUNCTION_IOC_INIT);

 921         ddi_put8(mpi2_dma_obj->acc_handle, &init->WhoInit,
 922             MPI2_WHOINIT_HOST_DRIVER);

 924         /* set MsgVersion and HeaderVersion host driver was built with */
 925         ddi_put16(mpi2_dma_obj->acc_handle, &init->MsgVersion,
 926             MPI2_VERSION);

 928         ddi_put16(mpi2_dma_obj->acc_handle, &init->HeaderVersion,
 929             MPI2_HEADER_VERSION);

 931         ddi_put16(mpi2_dma_obj->acc_handle, &init->SystemRequestFrameSize,
 932             instance->raid_io_msg_size / 4);

 934         ddi_put16(mpi2_dma_obj->acc_handle, &init->ReplyFreeQueueDepth,
 935             0);

 937         ddi_put16(mpi2_dma_obj->acc_handle,
 938             &init->ReplyDescriptorPostQueueDepth,
 939             instance->reply_q_depth);
 940         /*
 941          * These addresses are set using the DMA cookie addresses from when the
 942          * memory was allocated.  Sense buffer hi address should be 0.
 943          * ddi_put32(accessp, &init->SenseBufferAddressHigh, 0);
 944          */

 946         ddi_put32(mpi2_dma_obj->acc_handle,
```



```
 947             &init->SenseBufferAddressHigh, 0);

 949         ddi_put64(mpi2_dma_obj->acc_handle,
 950             (uint64_t *)&init->SystemRequestFrameBaseAddress,
 951             instance->io_request_frames_phy);

 953         ddi_put64(mpi2_dma_obj->acc_handle,
 954             &init->ReplyDescriptorPostQueueAddress,
 955             instance->reply_frame_pool_phy);

 957         ddi_put64(mpi2_dma_obj->acc_handle,
 958             &init->ReplyFreeQueueAddress, 0);

 960         cmd = instance->cmd_list[0];
 961         if (cmd == NULL) {
 962                 return (DDI_FAILURE);
 963         }
 964         cmd->retry_count_for_ocr = 0;
 965         cmd->pkt = NULL;
 966         cmd->drv_pkt_time = 0;

 968         mfiFrameInit2 = (struct mrsas_init_frame2 *)cmd->scsi_io_request;
 969         con_log(CL_ANN1, (CE_CONT, "[mfi vaddr]%p", (void *)mfiFrameInit2));

 971         frame_hdr = &cmd->frame->hdr;

 973         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
 974             MFI_CMD_STATUS_POLL_MODE);

 976         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);

 978         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;

 980         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

 982         con_log(CL_ANN, (CE_CONT,
 983             "mrsas_tbolt_ioc_init: SMID:%x\n", cmd->SMID));

 985         /* Init the MFI Header */
 986         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
 987             &mfiFrameInit2->cmd, MFI_CMD_OP_INIT);

 989         con_log(CL_ANN1, (CE_CONT, "[CMD]%x", mfiFrameInit2->cmd));

 991         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
 992             &mfiFrameInit2->cmd_status,
 993             MFI_STAT_INVALID_STATUS);

 995         con_log(CL_ANN1, (CE_CONT, "[Status]%x", mfiFrameInit2->cmd_status));

 997         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
 998             &mfiFrameInit2->queue_info_new_phys_addr_lo,
 999             mpi2_dma_obj->dma_cookie[0].dmac_address);

1001         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1002             &mfiFrameInit2->data_xfer_len,
1003             sizeof (Mpi2IOCInitRequest_t));

1005         con_log(CL_ANN1, (CE_CONT, "[reply q desc addr]%x",
1006             (int)init->ReplyDescriptorPostQueueAddress));

1008         /* fill driver version information */
1009         fill_up_drv_ver(&drv_ver_info);

1011         /* allocate the driver version data transfer buffer */
1012         instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
```

```
1013         instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
1014         instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
1015         instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
1016         instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
1017         instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;

1019         if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
1020             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
1021                 dev_err(instance->dip, CE_WARN,
1022                     "fusion init: Could not allocate driver version buffer.");
1023                 return (DDI_FAILURE);
1024         }
1025         /* copy driver version to dma buffer */
1026         bzero(instance->drv_ver_dma_obj.buffer, sizeof (drv_ver_info.drv_ver));
1027         ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
1028             (uint8_t *)drv_ver_info.drv_ver,
1029             (uint8_t *)instance->drv_ver_dma_obj.buffer,
1030             sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);

1032         /* send driver version physical address to firmware */
1033         ddi_put64(cmd->frame_dma_obj.acc_handle, &mfiFrameInit2->driverversion,
1034             instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);

1036         con_log(CL_ANN1, (CE_CONT, "[MPIINIT2 frame Phys addr ]0x%x len = %x",
1037             mfiFrameInit2->queue_info_new_phys_addr_lo,
1038             (int)sizeof (Mpi2IOCInitRequest_t)));

1040         con_log(CL_ANN1, (CE_CONT, "[Length]%x", mfiFrameInit2->data_xfer_len));

1042         con_log(CL_ANN1, (CE_CONT, "[MFI frame Phys Address]%x len = %x",
1043             cmd->scsi_io_request_phys_addr,
1044             (int)sizeof (struct mrsas_init_frame2)));

1046         /* disable interrupts before sending INIT2 frame */
1047         instance->func_ptr->disable_intr(instance);

1049         req_desc.Words = cmd->scsi_io_request_phys_addr;
1050         req_desc.MFAIo.RequestFlags =
1051             (MPI2_REQ_DESCRIPT_FLAGS_MFA << MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1053         cmd->request_desc = &req_desc;

1055         /* issue the init frame */

1057         mutex_enter(&instance->reg_write_mtx);
1058         WR_IB_LOW_QPORT((uint32_t)(req_desc.Words), instance);
1059         WR_IB_HIGH_QPORT((uint32_t)(req_desc.Words >> 32), instance);
1060         mutex_exit(&instance->reg_write_mtx);

1062         con_log(CL_ANN1, (CE_CONT, "[cmd = %d] ", frame_hdr->cmd));
1063         con_log(CL_ANN1, (CE_CONT, "[cmd  Status= %x] ",
1064             frame_hdr->cmd_status));

1066         timeout = drv_sectohz(MFI_POLL_TIMEOUT_SECS);
1066         timeout = drv_usectohz(MFI_POLL_TIMEOUT_SECS * MICROSEC);
1067         do {
1068                 if (ddi_get8(cmd->frame_dma_obj.acc_handle,
1069                     &mfiFrameInit2->cmd_status) != MFI_CMD_STATUS_POLL_MODE)
1070                         break;
1071                 delay(1);
1072                 timeout--;
1073         } while (timeout > 0);

1075         if (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1076             &mfiFrameInit2->cmd_status) == 0) {
1077                 con_log(CL_ANN, (CE_NOTE, "INIT2 Success"));
```



```
1078         } else {
1079                 con_log(CL_ANN, (CE_WARN, "INIT2 Fail"));
1080                 mrsas_dump_reply_desc(instance);
1081                 goto fail_ioc_init;
1082         }

1084         mrsas_dump_reply_desc(instance);

1086         instance->unroll.verBuff = 1;

1088         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_ioc_init: SUCCESSFUL"));

1090         return (DDI_SUCCESS);


1093 fail_ioc_init:

1095         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

1097         return (DDI_FAILURE);
1098 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  161130 Wed Aug 19 07:25:09 2015
new/usr/src/uts/common/io/myri10ge/drv/myri10ge.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

2202 static void
2203 myri10ge_stop_locked(struct myri10ge_priv *mgp)
2204 {
2205         int status, old_down_cnt;
2206         myri10ge_cmd_t cmd;
2207         int wait_time = 10;
2208         int i, polling;

2210         old_down_cnt = mgp->down_cnt;
2211         mb();
2212         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_ETHERNET_DOWN, &cmd);
2213         if (status) {
2214                 cmn_err(CE_WARN, "%s: Couldn't bring down link\n", mgp->name);
2215         }

2217         while (old_down_cnt == *((volatile int *)&mgp->down_cnt)) {
2218                 delay(drv_sectohz(1));
2218                 delay(1 * drv_usectohz(1000000));
2219                 wait_time--;
2220                 if (wait_time == 0)
2221                         break;
2222         }
2223 again:
2224         if (old_down_cnt == *((volatile int *)&mgp->down_cnt)) {
2225                 cmn_err(CE_WARN, "%s: didn't get down irq\n", mgp->name);
2226                 for (i = 0; i < mgp->num_slices; i++) {
2227                         /*
2228                          * take and release the rx lock to ensure
2229                          * that no interrupt thread is blocked
2230                          * elsewhere in the stack, preventing
2231                          * completion
2232                          */

2234                         mutex_enter(&mgp->ss[i].rx_lock);
2235                         printf("%s: slice %d rx irq idle\n",
2236                             mgp->name, i);
2237                         mutex_exit(&mgp->ss[i].rx_lock);

2239                         /* verify that the poll handler is inactive */
2240                         mutex_enter(&mgp->ss->poll_lock);
2241                         polling = mgp->ss->rx_polling;
2242                         mutex_exit(&mgp->ss->poll_lock);
2243                         if (polling) {
2244                                 printf("%s: slice %d is polling\n",
2245                                     mgp->name, i);
2246                                 delay(drv_sectohz(1));
2246                                 delay(1 * drv_usectohz(1000000));
2247                                 goto again;
2248                         }
2249                 }
2250                 delay(drv_sectohz(1));
2250                 delay(1 * drv_usectohz(1000000));
2251                 if (old_down_cnt == *((volatile int *)&mgp->down_cnt)) {
2252                         cmn_err(CE_WARN, "%s: Never got down irq\n", mgp->name);
2253                 }
2254         }

2256         for (i = 0; i < mgp->num_slices; i++)
2257                 myri10ge_teardown_slice(&mgp->ss[i]);
```



```
2259         if (mgp->toeplitz_hash_table != NULL) {
2260                 kmem_free(mgp->toeplitz_hash_table,
2261                     sizeof (uint32_t) * 12 * 256);
2262                 mgp->toeplitz_hash_table = NULL;
2263         }
2264         mgp->running = MYRI10GE_ETH_STOPPED;
2265 }
______unchanged_portion_omitted_

5811 static int
5812 myri10ge_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
5813 {

5815         struct myri10ge_priv *mgp;
5816         mac_register_t *macp, *omacp;
5817         ddi_acc_handle_t handle;
5818         uint32_t csr, hdr_offset;
5819         int status, span, link_width, max_read_request_4k;
5820         unsigned long bus_number, dev_number, func_number;
5821         size_t bytes;
5822         offset_t ss_offset;
5823         uint8_t vso;

5825         if (cmd == DDI_RESUME) {
5826                 return (myri10ge_resume(dip));
5827         }

5829         if (cmd != DDI_ATTACH)
5830                 return (DDI_FAILURE);
5831         if (pci_config_setup(dip, &handle) != DDI_SUCCESS)
5832                 return (DDI_FAILURE);

5834         /* enable busmater and io space access */
5835         csr = pci_config_get32(handle, PCI_CONF_COMM);
5836         pci_config_put32(handle, PCI_CONF_COMM,
5837             (csr |PCI_COMM_ME|PCI_COMM_MAE));
5838         status = myri10ge_read_pcie_link_width(handle, &link_width);
5839         if (status != 0) {
5840                 cmn_err(CE_WARN, "could not read link width!\n");
5841                 link_width = 0;
5842         }
5843         max_read_request_4k = !myri10ge_set_max_readreq(handle);
5844         status = myri10ge_find_cap(handle, &vso, PCI_CAP_ID_VS);
5845         if (status != 0)
5846                 goto abort_with_cfg_hdl;
5847         if ((omacp = mac_alloc(MAC_VERSION)) == NULL)
5848                 goto abort_with_cfg_hdl;
5849         /*
5850          * XXXX Hack: mac_register_t grows in newer kernels.  To be
5851          * able to write newer fields, such as m_margin, without
5852          * writing outside allocated memory, we allocate our own macp
5853          * and pass that to mac_register()
5854          */
5855         macp = kmem_zalloc(sizeof (*macp) * 8, KM_SLEEP);
5856         macp->m_version = omacp->m_version;

5858         if ((mgp = (struct myri10ge_priv *)
5859             kmem_zalloc(sizeof (*mgp), KM_SLEEP)) == NULL) {
5860                 goto abort_with_macinfo;
5861         }
5862         ddi_set_driver_private(dip, mgp);

5864         /* setup device name for log messages */
5865         (void) sprintf(mgp->name, "myri10ge%d", ddi_get_instance(dip));
```

```
5867         mutex_enter(&myri10ge_param_lock);
5868         myri10ge_get_props(dip);
5869         mgp->intr_coal_delay = myri10ge_intr_coal_delay;
5870         mgp->pause = myri10ge_flow_control;
5871         mutex_exit(&myri10ge_param_lock);

5873         mgp->max_read_request_4k = max_read_request_4k;
5874         mgp->pcie_link_width = link_width;
5875         mgp->running = MYRI10GE_ETH_STOPPED;
5876         mgp->vso = vso;
5877         mgp->dip = dip;
5878         mgp->cfg_hdl = handle;

5880         mgp->timer_ticks = drv_sectohz(5);
5880         mgp->timer_ticks = 5 * drv_usectohz(1000000); /* 5 seconds */
5881         myri10ge_test_physical(dip);

5883         /* allocate command page */
5884         bytes = sizeof (*mgp->cmd);
5885         mgp->cmd = (mcp_cmd_response_t *)
5886             (void *)myri10ge_dma_alloc(dip, bytes,
5887             &myri10ge_misc_dma_attr, &myri10ge_dev_access_attr,
5888             DDI_DMA_CONSISTENT, DDI_DMA_RDWR|DDI_DMA_CONSISTENT,
5889             &mgp->cmd_dma, 1, DDI_DMA_DONTWAIT);
5890         if (mgp->cmd == NULL)
5891                 goto abort_with_mgp;

5893         (void) myri10ge_reg_set(dip, &mgp->reg_set, &span, &bus_number,
5894             &dev_number, &func_number);
5895         if (myri10ge_verbose)
5896                 printf("%s at %ld:%ld:%ld attaching\n", mgp->name,
5897                     bus_number, dev_number, func_number);
5898         status = ddi_regs_map_setup(dip, mgp->reg_set, (caddr_t *)&mgp->sram,
5899             (offset_t)0, (offset_t)span,  &myri10ge_dev_access_attr,
5900             &mgp->io_handle);
5901         if (status != DDI_SUCCESS) {
5902                 cmn_err(CE_WARN, "%s: couldn't map memory space", mgp->name);
5903                 printf("%s: reg_set = %d, span = %d, status = %d",
5904                     mgp->name, mgp->reg_set, span, status);
5905                 goto abort_with_mgp;
5906         }

5908         hdr_offset = *(uint32_t *)(void*)(mgp->sram +  MCP_HEADER_PTR_OFFSET);
5909         hdr_offset = ntohl(hdr_offset) & 0xffffc;
5910         ss_offset = hdr_offset +
5911             offsetof(struct mcp_gen_header, string_specs);
5912         mgp->sram_size = ntohl(*(uint32_t *)(void*)(mgp->sram + ss_offset));
5913         myri10ge_pio_copy32(mgp->eeprom_strings,
5914             (uint32_t *)(void*)((char *)mgp->sram + mgp->sram_size),
5915             MYRI10GE_EEPROM_STRINGS_SIZE);
5916         (void) memset(mgp->eeprom_strings +
5917             MYRI10GE_EEPROM_STRINGS_SIZE - 2, 0, 2);

5919         status = myri10ge_read_mac_addr(mgp);
5920         if (status) {
5921                 goto abort_with_mapped;
5922         }

5924         status = myri10ge_select_firmware(mgp);
5925         if (status != 0) {
5926                 cmn_err(CE_WARN, "%s: failed to load firmware\n", mgp->name);
5927                 goto abort_with_mapped;
5928         }

5930         status = myri10ge_probe_slices(mgp);
5931         if (status != 0) {
```



```
5932                 cmn_err(CE_WARN, "%s: failed to probe slices\n", mgp->name);
5933                 goto abort_with_dummy_rdma;
5934         }

5936         status = myri10ge_alloc_slices(mgp);
5937         if (status != 0) {
5938                 cmn_err(CE_WARN, "%s: failed to alloc slices\n", mgp->name);
5939                 goto abort_with_dummy_rdma;
5940         }

5942         /* add the interrupt handler */
5943         status = myri10ge_add_intrs(mgp, 1);
5944         if (status != 0) {
5945                 cmn_err(CE_WARN, "%s: Failed to add interrupt\n",
5946                     mgp->name);
5947                 goto abort_with_slices;
5948         }

5950         /* now that we have an iblock_cookie, init the mutexes */
5951         mutex_init(&mgp->cmd_lock, NULL, MUTEX_DRIVER, mgp->icookie);
5952         mutex_init(&mgp->intrlock, NULL, MUTEX_DRIVER, mgp->icookie);


5955         status = myri10ge_nic_stat_init(mgp);
5956         if (status != DDI_SUCCESS)
5957                 goto abort_with_interrupts;
5958         status = myri10ge_info_init(mgp);
5959         if (status != DDI_SUCCESS)
5960                 goto abort_with_stats;

5962         /*
5963          *      Initialize  GLD state
5964          */

5966         macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
5967         macp->m_driver = mgp;
5968         macp->m_dip = dip;
5969         macp->m_src_addr = mgp->mac_addr;
5970         macp->m_callbacks = &myri10ge_m_callbacks;
5971         macp->m_min_sdu = 0;
5972         macp->m_max_sdu = myri10ge_mtu -
5973             (sizeof (struct ether_header) + MXGEFW_PAD + VLAN_TAGSZ);
5974 #ifdef SOLARIS_S11
5975         macp->m_margin = VLAN_TAGSZ;
5976 #endif
5977         macp->m_v12n = MAC_VIRT_LEVEL1;
5978         status = mac_register(macp, &mgp->mh);
5979         if (status != 0) {
5980                 cmn_err(CE_WARN, "%s: mac_register failed with %d\n",
5981                     mgp->name, status);
5982                 goto abort_with_info;
5983         }
5984         myri10ge_ndd_init(mgp);
5985         if (myri10ge_verbose)
5986                 printf("%s: %s, tx bndry %d, fw %s\n", mgp->name,
5987                     mgp->intr_type, mgp->tx_boundary, mgp->fw_name);
5988         mutex_enter(&myri10ge_param_lock);
5989         mgp->next = mgp_list;
5990         mgp_list = mgp;
5991         mutex_exit(&myri10ge_param_lock);
5992         kmem_free(macp, sizeof (*macp) * 8);
5993         mac_free(omacp);
5994         return (DDI_SUCCESS);

5996 abort_with_info:
5997         myri10ge_info_destroy(mgp);
```

```
5999 abort_with_stats:
6000         myri10ge_nic_stat_destroy(mgp);

6002 abort_with_interrupts:
6003         mutex_destroy(&mgp->cmd_lock);
6004         mutex_destroy(&mgp->intrlock);
6005         myri10ge_rem_intrs(mgp, 1);

6007 abort_with_slices:
6008         myri10ge_free_slices(mgp);

6010 abort_with_dummy_rdma:
6011         myri10ge_dummy_rdma(mgp, 0);

6013 abort_with_mapped:
6014         ddi_regs_map_free(&mgp->io_handle);

6016         myri10ge_dma_free(&mgp->cmd_dma);

6018 abort_with_mgp:
6019         kmem_free(mgp, sizeof (*mgp));

6021 abort_with_macinfo:
6022         kmem_free(macp, sizeof (*macp) * 8);
6023         mac_free(omacp);

6025 abort_with_cfg_hdl:
6026         pci_config_teardown(&handle);
6027         return (DDI_FAILURE);

6029 }
```

_____***unchanged_portion_omitted_***

****
23456 Wed Aug 19 07:25:09 2015
new/usr/src/uts/common/io/net80211/net80211.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
 305 /*
 306  * Start Watchdog timer. After count down timer(s), ic_watchdog
 307  * will be called
 308  */
 309 void
 310 ieee80211_start_watchdog(ieee80211com_t *ic, uint32_t timer)
 311 {
 312         if (ic->ic_watchdog_timer == 0 && ic->ic_watchdog != NULL) {
 313                 ic->ic_watchdog_timer = timeout(ic->ic_watchdog, ic,
 314                     drv_sectohz(timer));
 314                     drv_usectohz(1000000 * timer));
 315         }
 316 }
```

_____unchanged_portion_omitted_

****
74304 Wed Aug 19 07:25:09 2015
new/usr/src/uts/common/io/nge/nge_main.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
2186 /*
2187  * Get/Release semaphore of SMU
2188  * For SMU enabled chipset
2189  * When nge driver is attached, driver should acquire
2190  * semaphore before PHY init and accessing MAC registers.
2191  * When nge driver is unattached, driver should release
2192  * semaphore.
2193  */

2195 static int
2196 nge_smu_sema(nge_t *ngep, boolean_t acquire)
2197 {
2198         nge_tx_en tx_en;
2199         uint32_t tries;

2201         if (acquire) {
2202                 for (tries = 0; tries < 5; tries++) {
2203                         tx_en.val = nge_reg_get32(ngep, NGE_TX_EN);
2204                         if (tx_en.bits.smu2mac == NGE_SMU_FREE)
2205                                 break;
2206                         delay(drv_sectohz(1));
2206                         delay(drv_usectohz(1000000));
2207                 }
2208                 if (tx_en.bits.smu2mac != NGE_SMU_FREE)
2209                         return (DDI_FAILURE);
2210                 for (tries = 0; tries < 5; tries++) {
2211                         tx_en.val = nge_reg_get32(ngep, NGE_TX_EN);
2212                         tx_en.bits.mac2smu = NGE_SMU_GET;
2213                         nge_reg_put32(ngep, NGE_TX_EN, tx_en.val);
2214                         tx_en.val = nge_reg_get32(ngep, NGE_TX_EN);

2216                         if (tx_en.bits.mac2smu == NGE_SMU_GET &&
2217                             tx_en.bits.smu2mac == NGE_SMU_FREE)
2218                                 return (DDI_SUCCESS);
2219                         drv_usecwait(10);
2220                 }
2221                 return (DDI_FAILURE);
2222         } else
2223                 nge_reg_put32(ngep, NGE_TX_EN, 0x0);

2225         return (DDI_SUCCESS);

2227 }
```

_____unchanged_portion_omitted_

****
66815 Wed Aug 19 07:25:09 2015
new/usr/src/uts/common/io/ntxn/unm_nic_main.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
1150 static void
1151 unm_watchdog(unsigned long v)
1152 {
1153         unm_adapter *adapter = (unm_adapter *)v;

1155         if ((adapter->portnum == 0) && unm_nic_check_temp(adapter)) {
1156                 /*
1157                  * We return without turning on the netdev queue as there
1158                  * was an overheated device
1159                  */
1160                 return;
1161         }

1163         unm_nic_handle_phy_intr(adapter);

1165         /*
1166          * This function schedules a call for itself.
1167          */
1168         adapter->watchdog_timer = timeout((void (*)(void *))&unm_watchdog,
1169             (void *)adapter, drv_sectohz(2));
1169             (void *)adapter, 2 * drv_usectohz(1000000));

1171 }
```

_____***unchanged_portion_omitted_***

****
56873 Wed Aug 19 07:25:10 2015
new/usr/src/uts/common/io/nxge/nxge_hio.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
783 /*
784  * ----------------------------------------------------------------------
785  * These are HIO debugging functions.
786  * ----------------------------------------------------------------------
787  */

789 /*
790  * nxge_delay
791  *
792  *      Delay <seconds> number of seconds.
793  *
794  * Arguments:
795  *      nxge
796  *      group   The group to append to
797  *      dc      The DMA channel to append
798  *
799  * Notes:
800  *      This is a developer-only function.
801  *
802  * Context:
803  *      Any domain
804  */
805 void
806 nxge_delay(
807         int seconds)
808 {
809         delay(drv_sectohz(seconds));
809         delay(drv_usectohz(seconds * 1000000));
810 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  182954 Wed Aug 19 07:25:10 2015
new/usr/src/uts/common/io/pcic.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

5434 int pcic_debounce_cnt = PCIC_REM_DEBOUNCE_CNT;
5435 int pcic_debounce_intr_time = PCIC_REM_DEBOUNCE_TIME;
5436 int pcic_debounce_cnt_ok = PCIC_DEBOUNCE_OK_CNT;

5438 #ifdef CARDBUS
5439 static uint32_t pcic_cbps_on = 0;
5440 static uint32_t pcic_cbps_off = CB_PS_NOTACARD | CB_PS_CCDMASK |
5441                                 CB_PS_XVCARD | CB_PS_YVCARD;
5442 #else
5443 static uint32_t pcic_cbps_on = CB_PS_16BITCARD;
5444 static uint32_t pcic_cbps_off = CB_PS_NOTACARD | CB_PS_CCDMASK |
5445                                 CB_PS_CBCARD |
5446                                 CB_PS_XVCARD | CB_PS_YVCARD;
5447 #endif
5448 static void
5449 pcic_handle_cd_change(pcicdev_t *pcic, pcic_socket_t *sockp, uint8_t status)
5450 {
5451         boolean_t       do_debounce = B_FALSE;
5452         int             debounce_time = drv_usectohz(pcic_debounce_time);
5453         uint8_t         irq;
5454         timeout_id_t    debounce;

5456         /*
5457          * Always reset debounce but may need to check original state later.
5458          */
5459         debounce = sockp->pcs_debounce_id;
5460         sockp->pcs_debounce_id = 0;

5462         /*
5463          * Check to see whether a card is present or not. There are
5464          *      only two states that we are concerned with - the state
5465          *      where both CD pins are asserted, which means that the
5466          *      card is fully seated, and the state where neither CD
5467          *      pin is asserted, which means that the card is not
5468          *      present.
5469          * The CD signals are generally very noisy and cause a lot of
5470          *      contact bounce as the card is being inserted and
5471          *      removed, so we need to do some software debouncing.
5472          */

5474 #ifdef PCIC_DEBUG
5475                 pcic_err(pcic->dip, 6,
5476                     "pcic%d handle_cd_change: socket %d card status 0x%x"
5477                     " deb 0x%p\n", ddi_get_instance(pcic->dip),
5478                     sockp->pcs_socket, status, debounce);
5479 #endif
5480         switch (status & PCIC_ISTAT_CD_MASK) {
5481         case PCIC_CD_PRESENT_OK:
5482                 sockp->pcs_flags &= ~(PCS_CARD_REMOVED|PCS_CARD_CBREM);
5483                 if (!(sockp->pcs_flags & PCS_CARD_PRESENT)) {
5484                 uint32_t cbps;
5485 #ifdef PCIC_DEBUG
5486                 pcic_err(pcic->dip, 8, "New card (0x%x)\n", sockp->pcs_flags);
5487 #endif
5488                 cbps = pcic_getcb(pcic, CB_PRESENT_STATE);
5489 #ifdef PCIC_DEBUG
5490                 pcic_err(pcic->dip, 8, "CBus PS (0x%x)\n", cbps);
5491 #endif
5492                 /*
```



```
5493                  * Check the CB bits are sane.
5494                  */
5495                 if ((cbps & pcic_cbps_on) != pcic_cbps_on ||
5496                     cbps & pcic_cbps_off) {
5497                         cmn_err(CE_WARN,
5498                             "%s%d: Odd Cardbus Present State 0x%x\n",
5499                             ddi_get_name(pcic->dip),
5500                             ddi_get_instance(pcic->dip),
5501                             cbps);
5502                         pcic_putcb(pcic, CB_EVENT_FORCE, CB_EF_CVTEST);
5503                         debounce = 0;
5504                         debounce_time = drv_sectohz(1);
5504                         debounce_time = drv_usectohz(1000000);
5505                 }
5506                 if (debounce) {
5507                         sockp->pcs_flags |= PCS_CARD_PRESENT;
5508                         if (pcic_do_insertion) {

5510                                 cbps = pcic_getcb(pcic, CB_PRESENT_STATE);

5512                                 if (cbps & CB_PS_16BITCARD) {
5513                                         pcic_err(pcic->dip,
5514                                             8, "16 bit card inserted\n");
5515                                         sockp->pcs_flags |= PCS_CARD_IS16BIT;
5516                                         /* calls pcm_adapter_callback() */
5517                                         if (pcic->pc_callback) {

5519                                                 (void) ddi_prop_update_string(
5520                                                     DDI_DEV_T_NONE,
5521                                                     pcic->dip, PCM_DEVICETYPE,
5522                                                     "pccard");
5523                                                 PC_CALLBACK(pcic->dip,
5524                                                     pcic->pc_cb_arg,
5525                                                     PCE_CARD_INSERT,
5526                                                     sockp->pcs_socket);
5527                                         }
5528                                 } else if (cbps & CB_PS_CBCARD) {
5529                                         pcic_err(pcic->dip,
5530                                             8, "32 bit card inserted\n");

5532                                         if (pcic->pc_flags & PCF_CARDBUS) {
5533                                                 sockp->pcs_flags |=
5534                                                     PCS_CARD_ISCARDBUS;
5535 #ifdef CARDBUS
5536                                                 if (!pcic_load_cardbus(pcic,
5537                                                     sockp)) {
5538                                                         pcic_unload_cardbus(
5539                                                             pcic, sockp);
5540                                                 }

5542 #else
5543                                                 cmn_err(CE_NOTE,
5544                                                     "32 bit Cardbus not"
5545                                                     " supported in"
5546                                                     " this device driver\n");
5547 #endif
5548                                         } else {
5549                                                 /*
5550                                                  * Ignore the card
5551                                                  */
5552                                                 cmn_err(CE_NOTE,
5553                                                     "32 bit Cardbus not"
5554                                                     " supported on this"
5555                                                     " device\n");
5556                                         }
5557                                 } else {
```

```
5558                                        cmn_err(CE_NOTE,
5559                                            "Unsupported PCMCIA card"
5560                                            " inserted\n");
5561                                }
5562                        }
5563                } else {
5564                        do_debounce = B_TRUE;
5565                }
5566                } else {
5567                /*
5568                 * It is possible to come through here if the system
5569                 * starts up with cards already inserted. Do nothing
5570                 * and don't worry about it.
5571                 */
5572 #ifdef PCIC_DEBUG
5573                pcic_err(pcic->dip, 5,
5574                    "pcic%d: Odd card insertion indication on socket %d\n",
5575                    ddi_get_instance(pcic->dip),
5576                    sockp->pcs_socket);
5577 #endif
5578                }
5579                break;

5581        default:
5582                if (!(sockp->pcs_flags & PCS_CARD_PRESENT)) {
5583                /*
5584                 * Someone has started to insert a card so delay a while.
5585                 */
5586                do_debounce = B_TRUE;
5587                break;
5588                }
5589                /*
5590                 * Otherwise this is basically the same as not present
5591                 * so fall through.
5592                 */

5594                /* FALLTHRU */
5595        case 0:
5596                if (sockp->pcs_flags & PCS_CARD_PRESENT) {
5597                        if (pcic->pc_flags & PCF_CBPWRCTL) {
5598                                pcic_putcb(pcic, CB_CONTROL, 0);
5599                        } else {
5600                                pcic_putb(pcic, sockp->pcs_socket,
5601                                    PCIC_POWER_CONTROL, 0);
5602                        (void) pcic_getb(pcic, sockp->pcs_socket,
5603                            PCIC_POWER_CONTROL);
5604                }
5605 #ifdef PCIC_DEBUG
5606                pcic_err(pcic->dip, 8, "Card removed\n");
5607 #endif
5608                sockp->pcs_flags &= ~PCS_CARD_PRESENT;

5610                if (sockp->pcs_flags & PCS_CARD_IS16BIT) {
5611                        sockp->pcs_flags &= ~PCS_CARD_IS16BIT;
5612                        if (pcic_do_removal && pcic->pc_callback) {
5613                                PC_CALLBACK(pcic->dip, pcic->pc_cb_arg,
5614                                    PCE_CARD_REMOVAL, sockp->pcs_socket);
5615                        }
5616                }
5617                if (sockp->pcs_flags & PCS_CARD_ISCARDBUS) {
5618                        sockp->pcs_flags &= ~PCS_CARD_ISCARDBUS;
5619                        sockp->pcs_flags |= PCS_CARD_CBREM;
5620                }
5621                sockp->pcs_flags |= PCS_CARD_REMOVED;

5623                do_debounce = B_TRUE;
```



```
5624                }
5625                if (debounce && (sockp->pcs_flags & PCS_CARD_REMOVED)) {
5626                        if (sockp->pcs_flags & PCS_CARD_CBREM) {
5627                /*
5628                 * Ensure that we do the unloading in the
5629                 * debounce handler, that way we're not doing
5630                 * nasty things in an interrupt handler. e.g.
5631                 * a USB device will wait for data which will
5632                 * obviously never come because we've
5633                 * unplugged the device, but the wait will
5634                 * wait forever because no interrupts can
5635                 * come in...
5636                 */
5637 #ifdef CARDBUS
5638                        pcic_unload_cardbus(pcic, sockp);
5639                        /* pcic_dump_all(pcic); */
5640 #endif
5641                        sockp->pcs_flags &= ~PCS_CARD_CBREM;
5642                        }
5643                        sockp->pcs_flags &= ~PCS_CARD_REMOVED;
5644                }
5645                break;
5646        } /* switch */

5648        if (do_debounce) {
5649        /*
5650         * Delay doing
5651         * anything for a while so that things can settle
5652         * down a little. Interrupts are already disabled.
5653         * Reset the state and we'll reevaluate the
5654         * whole kit 'n kaboodle when the timeout fires
5655         */
5656 #ifdef PCIC_DEBUG
5657                pcic_err(pcic->dip, 8, "Queueing up debounce timeout for "
5658                    "socket %d.%d\n",
5659                    ddi_get_instance(pcic->dip),
5660                    sockp->pcs_socket);
5661 #endif
5662                sockp->pcs_debounce_id =
5663                    pcic_add_debqueue(sockp, debounce_time);

5665        /*
5666         * We bug out here without re-enabling interrupts. They will
5667         * be re-enabled when the debounce timeout swings through
5668         * here.
5669         */
5670                return;
5671        }

5673        /*
5674         * Turn on Card detect interrupts. Other interrupts will be
5675         * enabled during set_socket calls.
5676         *
5677         * Note that set_socket only changes interrupt settings when there
5678         * is a card present.
5679         */
5680        irq = pcic_getb(pcic, sockp->pcs_socket, PCIC_MANAGEMENT_INT);
5681        irq |= PCIC_CD_DETECT;
5682        pcic_putb(pcic, sockp->pcs_socket, PCIC_MANAGEMENT_INT, irq);
5683        pcic_putcb(pcic, CB_STATUS_MASK, CB_SE_CCDMASK);

5685        /* Out from debouncing state */
5686        sockp->pcs_flags &= ~PCS_DEBOUNCING;

5688        pcic_err(pcic->dip, 7, "Leaving pcic_handle_cd_change\n");
5689 }
```

unchanged_portion_omitted_

****
65050 Wed Aug 19 07:25:10 2015
new/usr/src/uts/common/io/pciex/hotplug/pciehpc.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
1034 /*
1035  * pciehpc_slot_poweron()
1036  *
1037  * Poweron/Enable the slot.
1038  *
1039  * Note: This function is called by DDI HP framework at kernel context only
1040  */
1041 /*ARGSUSED*/
1042 static int
1043 pciehpc_slot_poweron(pcie_hp_slot_t *slot_p, ddi_hp_cn_state_t *result)
1044 {
1045         pcie_hp_ctrl_t  *ctrl_p = slot_p->hs_ctrl;
1046         pcie_bus_t      *bus_p = PCIE_DIP2BUS(ctrl_p->hc_dip);
1047         uint16_t        status, control;

1049         ASSERT(MUTEX_HELD(&ctrl_p->hc_mutex));

1051         /* get the current state of the slot */
1052         pciehpc_get_slot_state(slot_p);

1054         /* check if the slot is already in the 'enabled' state */
1055         if (slot_p->hs_info.cn_state >= DDI_HP_CN_STATE_POWERED) {
1056                 /* slot is already in the 'enabled' state */
1057                 PCIE_DBG("pciehpc_slot_poweron() slot %d already enabled\n",
1058                     slot_p->hs_phy_slot_num);

1060                 *result = slot_p->hs_info.cn_state;
1061                 return (DDI_SUCCESS);
1062         }

1064         /* read the Slot Status Register */
1065         status =  pciehpc_reg_get16(ctrl_p,
1066             bus_p->bus_pcie_off + PCIE_SLOTSTS);

1068         /* make sure the MRL switch is closed if present */
1069         if ((ctrl_p->hc_has_mrl) && (status & PCIE_SLOTSTS_MRL_SENSOR_OPEN)) {
1070                 /* MRL switch is open */
1071                 cmn_err(CE_WARN, "MRL switch is open on slot %d\n",
1072                     slot_p->hs_phy_slot_num);
1073                 goto cleanup;
1074         }

1076         /* make sure the slot has a device present */
1077         if (!(status & PCIE_SLOTSTS_PRESENCE_DETECTED)) {
1078                 /* slot is empty */
1079                 PCIE_DBG("slot %d is empty\n", slot_p->hs_phy_slot_num);
1080                 goto cleanup;
1081         }

1083         /* get the current state of Slot Control Register */
1084         control =  pciehpc_reg_get16(ctrl_p,
1085             bus_p->bus_pcie_off + PCIE_SLOTCTL);

1087         /*
1088          * Enable power to the slot involves:
1089          *      1. Set power LED to blink and ATTN led to OFF.
1090          *      2. Set power control ON in Slot Control Reigster and
1091          *         wait for Command Completed Interrupt or 1 sec timeout.
1092          *      3. If Data Link Layer State Changed events are supported
```



```
1093          *         then wait for the event to indicate Data Layer Link
1094          *         is active. The time out value for this event is 1 second.
1095          *         This is specified in PCI-E version 1.1.
1096          *      4. Set power LED to be ON.
1097          */

1099         /* 1. set power LED to blink & ATTN led to OFF */
1100         pciehpc_set_led_state(ctrl_p, PCIE_HP_POWER_LED, PCIE_HP_LED_BLINK);
1101         pciehpc_set_led_state(ctrl_p, PCIE_HP_ATTN_LED, PCIE_HP_LED_OFF);

1103         /* 2. set power control to ON */
1104         control =  pciehpc_reg_get16(ctrl_p,
1105             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1106         control &= ~PCIE_SLOTCTL_PWR_CONTROL;
1107         pciehpc_issue_hpc_command(ctrl_p, control);

1109         /* 3. wait for DLL State Change event, if it's supported */
1110         if (ctrl_p->hc_dll_active_rep) {
1111                 status =  pciehpc_reg_get16(ctrl_p,
1112                     bus_p->bus_pcie_off + PCIE_LINKSTS);

1114                 if (!(status & PCIE_LINKSTS_DLL_LINK_ACTIVE)) {
1115                         /* wait 1 sec for the DLL State Changed event */
1116                         (void) cv_timedwait(&slot_p->hs_dll_active_cv,
1117                             &ctrl_p->hc_mutex,
1118                             ddi_get_lbolt() +
1119                             SEC_TO_TICK(PCIE_HP_DLL_STATE_CHANGE_TIMEOUT));

1121                         /* check Link status */
1122                         status =  pciehpc_reg_get16(ctrl_p,
1123                             bus_p->bus_pcie_off +
1124                             PCIE_LINKSTS);
1125                         if (!(status & PCIE_LINKSTS_DLL_LINK_ACTIVE))
1126                                 goto cleanup2;
1127                 }
1128         }

1130         /* wait 1 sec for link to come up */
1131         delay(drv_sectohz(1));
1131         delay(drv_usectohz(1000000));

1133         /* check power is really turned ON */
1134         control =  pciehpc_reg_get16(ctrl_p,
1135             bus_p->bus_pcie_off + PCIE_SLOTCTL);

1137         if (control & PCIE_SLOTCTL_PWR_CONTROL) {
1138                 PCIE_DBG("slot %d fails to turn on power on connect\n",
1139                     slot_p->hs_phy_slot_num);

1141                 goto cleanup1;
1142         }

1144         /* clear power fault status */
1145         status =  pciehpc_reg_get16(ctrl_p,
1146             bus_p->bus_pcie_off + PCIE_SLOTSTS);
1147         status |= PCIE_SLOTSTS_PWR_FAULT_DETECTED;
1148         pciehpc_reg_put16(ctrl_p, bus_p->bus_pcie_off + PCIE_SLOTSTS,
1149             status);

1151         /* enable power fault detection interrupt */
1152         control |= PCIE_SLOTCTL_PWR_FAULT_EN;
1153         pciehpc_issue_hpc_command(ctrl_p, control);

1155         /* 4. Set power LED to be ON */
1156         pciehpc_set_led_state(ctrl_p, PCIE_HP_POWER_LED, PCIE_HP_LED_ON);
```

```
1158         /* if EMI is present, turn it ON */
1159         if (ctrl_p->hc_has_emi_lock) {
1160                 status =  pciehpc_reg_get16(ctrl_p,
1161                     bus_p->bus_pcie_off + PCIE_SLOTSTS);

1163                 if (!(status & PCIE_SLOTSTS_EMI_LOCK_SET)) {
1164                         control =  pciehpc_reg_get16(ctrl_p,
1165                             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1166                         control |= PCIE_SLOTCTL_EMI_LOCK_CONTROL;
1167                         pciehpc_issue_hpc_command(ctrl_p, control);

1169                         /* wait 1 sec after toggling the state of EMI lock */
1170                         delay(drv_sectohz(1));
1170                         delay(drv_usectohz(1000000));
1171                 }
1172         }

1174         *result = slot_p->hs_info.cn_state =
1175             DDI_HP_CN_STATE_POWERED;

1177         return (DDI_SUCCESS);

1179 cleanup2:
1180         control =  pciehpc_reg_get16(ctrl_p,
1181             bus_p->bus_pcie_off + PCIE_SLOTCTL);

1183         /* if power is ON, set power control to OFF */
1184         if (!(control & PCIE_SLOTCTL_PWR_CONTROL)) {
1185                 control |= PCIE_SLOTCTL_PWR_CONTROL;
1186                 pciehpc_issue_hpc_command(ctrl_p, control);
1187         }

1189 cleanup1:
1190         /* set power led to OFF */
1191         pciehpc_set_led_state(ctrl_p, PCIE_HP_POWER_LED, PCIE_HP_LED_OFF);

1193 cleanup:
1194         return (DDI_FAILURE);
1195 }

1197 /*ARGSUSED*/
1198 static int
1199 pciehpc_slot_poweroff(pcie_hp_slot_t *slot_p, ddi_hp_cn_state_t *result)
1200 {
1201         pcie_hp_ctrl_t  *ctrl_p = slot_p->hs_ctrl;
1202         pcie_bus_t      *bus_p = PCIE_DIP2BUS(ctrl_p->hc_dip);
1203         uint16_t        status, control;

1205         ASSERT(MUTEX_HELD(&ctrl_p->hc_mutex));

1207         /* get the current state of the slot */
1208         pciehpc_get_slot_state(slot_p);

1210         /* check if the slot is not in the "enabled' state */
1211         if (slot_p->hs_info.cn_state < DDI_HP_CN_STATE_POWERED) {
1212                 /* slot is in the 'disabled' state */
1213                 PCIE_DBG("pciehpc_slot_poweroff(): "
1214                     "slot %d already disabled\n", slot_p->hs_phy_slot_num);
1215                 ASSERT(slot_p->hs_power_led_state == PCIE_HP_LED_OFF);

1217                 *result = slot_p->hs_info.cn_state;
1218                 return (DDI_SUCCESS);
1219         }

1221         /* read the Slot Status Register */
1222         status =  pciehpc_reg_get16(ctrl_p,
```



```
1223             bus_p->bus_pcie_off + PCIE_SLOTSTS);

1225         /* make sure the slot has a device present */
1226         if (!(status & PCIE_SLOTSTS_PRESENCE_DETECTED)) {
1227                 /* slot is empty */
1228                 PCIE_DBG("pciehpc_slot_poweroff(): slot %d is empty\n",
1229                     slot_p->hs_phy_slot_num);
1230                 goto cleanup;
1231         }

1233         /*
1234          * Disable power to the slot involves:
1235          *      1. Set power LED to blink.
1236          *      2. Set power control OFF in Slot Control Reigster and
1237          *         wait for Command Completed Interrupt or 1 sec timeout.
1238          *      3. Set POWER led and ATTN led to be OFF.
1239          */

1241         /* 1. set power LED to blink */
1242         pciehpc_set_led_state(ctrl_p, PCIE_HP_POWER_LED, PCIE_HP_LED_BLINK);

1244         /* disable power fault detection interrupt */
1245         control = pciehpc_reg_get16(ctrl_p,
1246             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1247         control &= ~PCIE_SLOTCTL_PWR_FAULT_EN;
1248         pciehpc_issue_hpc_command(ctrl_p, control);

1250         /* 2. set power control to OFF */
1251         control =  pciehpc_reg_get16(ctrl_p,
1252             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1253         control |= PCIE_SLOTCTL_PWR_CONTROL;
1254         pciehpc_issue_hpc_command(ctrl_p, control);

1256 #ifdef DEBUG
1257         /* check for power control bit to be OFF */
1258         control =  pciehpc_reg_get16(ctrl_p,
1259             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1260         ASSERT(control & PCIE_SLOTCTL_PWR_CONTROL);
1261 #endif

1263         /* 3. Set power LED to be OFF */
1264         pciehpc_set_led_state(ctrl_p, PCIE_HP_POWER_LED, PCIE_HP_LED_OFF);
1265         pciehpc_set_led_state(ctrl_p, PCIE_HP_ATTN_LED, PCIE_HP_LED_OFF);

1267         /* if EMI is present, turn it OFF */
1268         if (ctrl_p->hc_has_emi_lock) {
1269                 status =  pciehpc_reg_get16(ctrl_p,
1270                     bus_p->bus_pcie_off + PCIE_SLOTSTS);

1272                 if (status & PCIE_SLOTSTS_EMI_LOCK_SET) {
1273                         control =  pciehpc_reg_get16(ctrl_p,
1274                             bus_p->bus_pcie_off + PCIE_SLOTCTL);
1275                         control |= PCIE_SLOTCTL_EMI_LOCK_CONTROL;
1276                         pciehpc_issue_hpc_command(ctrl_p, control);

1278                         /* wait 1 sec after toggling the state of EMI lock */
1279                         delay(drv_sectohz(1));
1279                         delay(drv_usectohz(1000000));
1280                 }
1281         }

1283         /* get the current state of the slot */
1284         pciehpc_get_slot_state(slot_p);

1286         *result = slot_p->hs_info.cn_state;
```

```
1288         return (DDI_SUCCESS);

1290 cleanup:
1291         return (DDI_FAILURE);
1292 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   99476 Wed Aug 19 07:25:10 2015
new/usr/src/uts/common/io/pshot.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3033 /*
3034  * Handle some special cases for testing bus_config via pshot
3035  *
3036  * Match these special address formats to behavior:
3037  *
3038  *      err.*           - induce bus_config error
3039  *      delay           - induce 1 second of bus_config delay time
3040  *      delay,n         - induce n seconds of bus_config delay time
3041  *      wait            - induce 1 second of bus_config wait time
3042  *      wait,n          - induce n seconds of bus_config wait time
3043  *      failinit.*      - induce error at INITCHILD
3044  *      failprobe.*     - induce error at probe
3045  *      failattach.*    - induce error at attach
3046  */
3047 /*ARGSUSED*/
3048 static int
3049 pshot_bus_config_test_specials(dev_info_t *parent, char *devname,
3050         char *cname, char *caddr)
3051 {
3052         char    *p;
3053         int     n;

3055         if (strncmp(caddr, "err", 3) == 0) {
3056                 if (pshot_debug)
3057                         cmn_err(CE_CONT,
3058                             "pshot%d: %s forced failure\n",
3059                             ddi_get_instance(parent), devname);
3060                 return (NDI_FAILURE);
3061         }

3063         /*
3064          * The delay and wait strings have the same effect.
3065          * The "wait[,]" support should be removed once the
3066          * devfs test suites are fixed.
3067          * NOTE: delay should not be called from interrupt context
3068          */
3069         ASSERT(!servicing_interrupt());

3071         if (strncmp(caddr, "delay,", 6) == 0) {
3072                 p = caddr+6;
3073                 n = stoi(&p);
3074                 if (*p != 0)
3075                         n = 1;
3076                 if (pshot_debug)
3077                         cmn_err(CE_CONT,
3078                             "pshot%d: %s delay %d second\n",
3079                             ddi_get_instance(parent), devname, n);
3080                 delay(drv_sectohz(n));
3080                 delay(n * drv_usectohz(1000000));
3081         } else if (strncmp(caddr, "delay", 5) == 0) {
3082                 if (pshot_debug)
3083                         cmn_err(CE_CONT,
3084                             "pshot%d: %s delay 1 second\n",
3085                             ddi_get_instance(parent), devname);
3086                 delay(drv_sectohz(1));
3086                 delay(drv_usectohz(1000000));
3087         } else if (strncmp(caddr, "wait,", 5) == 0) {
3088                 p = caddr+5;
3089                 n = stoi(&p);
```



```
3090                 if (*p != 0)
3091                         n = 1;
3092                 if (pshot_debug)
3093                         cmn_err(CE_CONT,
3094                             "pshot%d: %s wait %d second\n",
3095                             ddi_get_instance(parent), devname, n);
3096                 delay(drv_sectohz(n));
3096                 delay(n * drv_usectohz(1000000));
3097         } else if (strncmp(caddr, "wait", 4) == 0) {
3098                 if (pshot_debug)
3099                         cmn_err(CE_CONT,
3100                             "pshot%d: %s wait 1 second\n",
3101                             ddi_get_instance(parent), devname);
3102                 delay(drv_sectohz(1));
3102                 delay(drv_usectohz(1000000));
3103         }

3105         return (NDI_SUCCESS);
3106 }
______unchanged_portion_omitted_

3470 static void
3471 pshot_walk_thread()
3472 {
3473         static void pshot_timeout(void *arg);
3474         static kthread_id_t pwt;

3476         pwt = curthread;
3477         mutex_init(&pwl, NULL, MUTEX_DRIVER, NULL);
3478         cv_init(&pwcv, NULL, CV_DRIVER, NULL);

3480         while (1) {
3481                 pshot_walk_tree();
3482                 mutex_enter(&pwl);
3483                 (void) timeout(pshot_timeout, NULL, drv_sectohz(5));
3483                 (void) timeout(pshot_timeout, NULL, 5 * drv_usectohz(1000000));
3484                 cv_wait(&pwcv, &pwl);
3485                 mutex_exit(&pwl);
3486         }
3487 }
______unchanged_portion_omitted_

3537 static void
3538 pshot_event_test(void *arg)
3539 {
3540         pshot_t *pshot = (pshot_t *)arg;
3541         ndi_event_hdl_t hdl;
3542         ndi_event_set_t events;
3543         int i, rval;

3545         (void) ndi_event_alloc_hdl(pshot->dip, NULL, &hdl, NDI_SLEEP);

3547         events.ndi_events_version = NDI_EVENTS_REV1;
3548         events.ndi_n_events = PSHOT_N_TEST_EVENTS;
3549         events.ndi_event_defs = pshot_test_events;

3551         cmn_err(CE_CONT, "pshot: binding set of 8 events\n");
3552         delay(drv_sectohz(1));
3552         delay(drv_usectohz(1000000));
3553         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3554         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3556         cmn_err(CE_CONT, "pshot: binding the same set of 8 events\n");
3557         delay(drv_sectohz(1));
3557         delay(drv_usectohz(1000000));
3558         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
```

```
3559         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3561         cmn_err(CE_CONT, "pshot: unbinding  all events\n");
3562         delay(drv_sectohz(1));
3562         delay(drv_usectohz(1000000));
3563         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3564         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);


3567         cmn_err(CE_CONT, "pshot: binding one highlevel event\n");
3568         delay(drv_sectohz(1));
3568         delay(drv_usectohz(1000000));
3569         events.ndi_n_events = 1;
3570         events.ndi_event_defs = pshot_test_events_high;
3571         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3572         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3574         cmn_err(CE_CONT, "pshot: binding the same set of 8 events\n");
3575         delay(drv_sectohz(1));
3575         delay(drv_usectohz(1000000));
3576         events.ndi_n_events = PSHOT_N_TEST_EVENTS;
3577         events.ndi_event_defs = pshot_test_events;
3578         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3579         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3581         cmn_err(CE_CONT, "pshot: unbinding one highlevel event\n");
3582         delay(drv_sectohz(1));
3582         delay(drv_usectohz(1000000));
3583         events.ndi_n_events = 1;
3584         events.ndi_event_defs = pshot_test_events_high;
3585         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3586         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3588         cmn_err(CE_CONT, "pshot: binding one highlevel event\n");
3589         delay(drv_sectohz(1));
3589         delay(drv_usectohz(1000000));
3590         events.ndi_n_events = 1;
3591         events.ndi_event_defs = pshot_test_events_high;
3592         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3593         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3595         cmn_err(CE_CONT, "pshot: unbinding one highlevel event\n");
3596         delay(drv_sectohz(1));
3596         delay(drv_usectohz(1000000));
3597         events.ndi_n_events = 1;
3598         events.ndi_event_defs = pshot_test_events_high;
3599         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3600         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3602         cmn_err(CE_CONT, "pshot: binding the same set of 8 events\n");
3603         delay(drv_sectohz(1));
3603         delay(drv_usectohz(1000000));
3604         events.ndi_n_events = PSHOT_N_TEST_EVENTS;
3605         events.ndi_event_defs = pshot_test_events;
3606         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3607         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3609         cmn_err(CE_CONT, "pshot: unbinding first 2 events\n");
3610         delay(drv_sectohz(1));
3610         delay(drv_usectohz(1000000));
3611         events.ndi_n_events = 2;
3612         events.ndi_event_defs = pshot_test_events;
3613         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3614         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3616         cmn_err(CE_CONT, "pshot: unbinding first 2 events again\n");
```



```
3617         delay(drv_sectohz(1));
3617         delay(drv_usectohz(1000000));
3618         events.ndi_n_events = 2;
3619         events.ndi_event_defs = pshot_test_events;
3620         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3621         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3623         cmn_err(CE_CONT, "pshot: unbinding  middle 2 events\n");
3624         delay(drv_sectohz(1));
3624         delay(drv_usectohz(1000000));
3625         events.ndi_n_events = 2;
3626         events.ndi_event_defs = &pshot_test_events[4];
3627         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3628         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3630         cmn_err(CE_CONT, "pshot: binding those 2 events back\n");
3631         delay(drv_sectohz(1));
3631         delay(drv_usectohz(1000000));
3632         events.ndi_n_events = 2;
3633         events.ndi_event_defs = &pshot_test_events[4];
3634         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3635         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3637         cmn_err(CE_CONT, "pshot: unbinding  2 events\n");
3638         delay(drv_sectohz(1));
3638         delay(drv_usectohz(1000000));
3639         events.ndi_n_events = 2;
3640         events.ndi_event_defs = &pshot_test_events[4];
3641         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3642         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3644         cmn_err(CE_CONT, "pshot: unbinding  all events\n");
3645         delay(drv_sectohz(1));
3645         delay(drv_usectohz(1000000));
3646         events.ndi_n_events = PSHOT_N_TEST_EVENTS;
3647         events.ndi_event_defs = pshot_test_events;
3648         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3649         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3651         cmn_err(CE_CONT, "pshot: unbinding  1 event\n");
3652         delay(drv_sectohz(1));
3652         delay(drv_usectohz(1000000));
3653         events.ndi_n_events = 1;
3654         events.ndi_event_defs = &pshot_test_events[2];
3655         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3656         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3658         cmn_err(CE_CONT, "pshot: unbinding  1 event\n");
3659         delay(drv_sectohz(1));
3659         delay(drv_usectohz(1000000));
3660         events.ndi_n_events = 1;
3661         events.ndi_event_defs = &pshot_test_events[3];
3662         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3663         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3665         cmn_err(CE_CONT, "pshot: unbinding  1 event\n");
3666         delay(drv_sectohz(1));
3666         delay(drv_usectohz(1000000));
3667         events.ndi_n_events = 1;
3668         events.ndi_event_defs = &pshot_test_events[6];
3669         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3670         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3672         cmn_err(CE_CONT, "pshot: unbinding  1 event\n");
3673         delay(drv_sectohz(1));
3673         delay(drv_usectohz(1000000));
```

```
3674         events.ndi_n_events = 1;
3675         events.ndi_event_defs = &pshot_test_events[7];
3676         rval = ndi_event_unbind_set(hdl, &events, NDI_SLEEP);
3677         cmn_err(CE_CONT, "pshot: ndi_event_unbind_set rval = %d\n", rval);

3679         events.ndi_n_events = PSHOT_N_TEST_EVENTS;
3680         events.ndi_event_defs = pshot_test_events;

3682         cmn_err(CE_CONT, "pshot: binding set of 8 events\n");
3683         delay(drv_sectohz(1));
3683         delay(drv_usectohz(1000000));
3684         rval = ndi_event_bind_set(hdl, &events, NDI_SLEEP);
3685         cmn_err(CE_CONT, "pshot: ndi_event_bind_set rval = %d\n", rval);

3687         cmn_err(CE_CONT, "pshot: adding 8 callbacks\n");
3688         delay(drv_sectohz(1));
3688         delay(drv_usectohz(1000000));
3689         for (i = 0; i < 8; i++) {
3690                 rval = ndi_event_add_callback(hdl, pshot->dip,
3691                     ndi_event_tag_to_cookie(hdl,
3692                     pshot_test_events[i].ndi_event_tag),
3693                     pshot_event_cb_test,
3694                     (void *)(uintptr_t)pshot_test_events[i].ndi_event_tag,
3695                     NDI_SLEEP, &pshot->test_callback_cache[i]);
3696                 ASSERT(rval == NDI_SUCCESS);
3697         }

3699         cmn_err(CE_CONT, "pshot: event callbacks\n");

3701         for (i = 10; i < 18; i++) {
3702                 ddi_eventcookie_t cookie = ndi_event_tag_to_cookie(hdl, i);

3704                 rval = ndi_event_run_callbacks(hdl, pshot->dip, cookie,
3705                     (void *)hdl);

3707                 cmn_err(CE_CONT, "pshot: callback, tag=%d rval=%d\n",
3708                     i, rval);
3709                 delay(drv_sectohz(1));
3709                 delay(drv_usectohz(1000000));
3710         }

3712         cmn_err(CE_CONT, "pshot: redo event callbacks\n");

3714         for (i = 10; i < 18; i++) {
3715                 ddi_eventcookie_t cookie = ndi_event_tag_to_cookie(hdl, i);

3717                 rval = ndi_event_run_callbacks(hdl,
3718                     pshot->dip, cookie, (void *)hdl);

3720                 cmn_err(CE_CONT, "pshot: callback, tag=%d rval=%d\n",
3721                     i, rval);
3722                 delay(drv_sectohz(1));
3722                 delay(drv_usectohz(1000000));
3723         }

3725         cmn_err(CE_CONT, "pshot: removing 8 callbacks\n");
3726         delay(drv_sectohz(1));
3726         delay(drv_usectohz(1000000));

3728         for (i = 0; i < 8; i++) {
3729                 (void) ndi_event_remove_callback(hdl,
3730                     pshot->test_callback_cache[i]);

3732                 pshot->test_callback_cache[i] = 0;
3733         }
```



```
3735         cmn_err(CE_CONT, "pshot: freeing handle with bound set\n");
3736         delay(drv_sectohz(1));
3736         delay(drv_usectohz(1000000));

3738         rval =  ndi_event_free_hdl(hdl);

3740         ASSERT(rval == NDI_SUCCESS);

3742 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   86759 Wed Aug 19 07:25:11 2015
new/usr/src/uts/common/io/rtw/rtw.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

2438 /*
2439  * Examine and potentially adjust the transmit rate.
2440  */
2441 static void
2442 rtw_rate_ctl(void *arg)
2443 {
2444         ieee80211com_t  *ic = (ieee80211com_t *)arg;
2445         rtw_softc_t *rsc = (rtw_softc_t *)ic;
2446         struct ieee80211_node *in = ic->ic_bss;
2447         struct ieee80211_rateset *rs = &in->in_rates;
2448         int32_t mod = 1, nrate, enough;

2450         mutex_enter(&rsc->sc_genlock);
2451         enough = (rsc->sc_tx_ok + rsc->sc_tx_err) >= 600? 1 : 0;

2453         /* err ratio is high -> down */
2454         if (enough && rsc->sc_tx_ok < rsc->sc_tx_err)
2455                 mod = -1;

2457         nrate = in->in_txrate;
2458         switch (mod) {
2459         case -1:
2460                 if (nrate > 0) {
2461                         nrate--;
2462                 }
2463                 break;
2464         case 1:
2465                 if (nrate + 1 < rs->ir_nrates) {
2466                         nrate++;
2467                 }
2468                 break;
2469         }

2471         if (nrate != in->in_txrate)
2472                 in->in_txrate = nrate;
2473         rsc->sc_tx_ok = rsc->sc_tx_err = rsc->sc_tx_retr = 0;
2474         mutex_exit(&rsc->sc_genlock);
2475         if (ic->ic_state == IEEE80211_S_RUN)
2476                 rsc->sc_ratectl_id = timeout(rtw_rate_ctl, ic,
2477                     drv_sectohz(1));
2477                     drv_usectohz(1000000));
2478 }

2480 static int32_t
2481 rtw_new_state(ieee80211com_t *ic, enum ieee80211_state nstate, int arg)
2482 {
2483         rtw_softc_t *rsc = (rtw_softc_t *)ic;
2484         int error;
2485         enum ieee80211_state ostate;

2487         ostate = ic->ic_state;

2489         RTW_DPRINTF(RTW_DEBUG_ATTACH,
2490             "rtw_new_state: ostate:0x%x, nstate:0x%x, opmode:0x%x\n",
2491             ostate, nstate, ic->ic_opmode);


2494         mutex_enter(&rsc->sc_genlock);
2495         if (rsc->sc_scan_id != 0) {
```



```
2496                 (void) untimeout(rsc->sc_scan_id);
2497                 rsc->sc_scan_id = 0;
2498         }
2499         if (rsc->sc_ratectl_id != 0) {
2500                 (void) untimeout(rsc->sc_ratectl_id);
2501                 rsc->sc_ratectl_id = 0;
2502         }
2503         rtw_rate_ctl_reset(rsc, nstate);
2504         if (ostate == IEEE80211_S_INIT && nstate != IEEE80211_S_INIT)
2505                 (void) rtw_pwrstate(rsc, RTW_ON);
2506         if (nstate != IEEE80211_S_INIT) {
2507                 if ((error = rtw_tune(rsc)) != 0) {
2508                         mutex_exit(&rsc->sc_genlock);
2509                         return (error);
2510                 }
2511         }
2512         switch (nstate) {
2513         case IEEE80211_S_INIT:
2514                 RTW_DPRINTF(RTW_DEBUG_ATTACH, "rtw_new_state: S_INIT\n");
2515                 break;
2516         case IEEE80211_S_SCAN:
2517                 RTW_DPRINTF(RTW_DEBUG_ATTACH, "rtw_new_state: S_SCAN\n");
2518                 rsc->sc_scan_id = timeout(rtw_next_scan, ic,
2519                     drv_usectohz(200000));
2520                 rtw_set_nettype(rsc, IEEE80211_M_MONITOR);
2521                 break;
2522         case IEEE80211_S_RUN:
2523                 RTW_DPRINTF(RTW_DEBUG_ATTACH, "rtw_new_state: S_RUN\n");
2524                 switch (ic->ic_opmode) {
2525                 case IEEE80211_M_HOSTAP:
2526                 case IEEE80211_M_IBSS:
2527                         rtw_set_nettype(rsc, IEEE80211_M_MONITOR);
2528                         /* TBD */
2529                         /*FALLTHROUGH*/
2530                 case IEEE80211_M_AHDEMO:
2531                 case IEEE80211_M_STA:
2532                         RTW_DPRINTF(RTW_DEBUG_ATTACH,
2533                             "rtw_new_state: sta\n");
2534                         rtw_join_bss(rsc, ic->ic_bss->in_bssid, 0);
2535                         rsc->sc_ratectl_id = timeout(rtw_rate_ctl, ic,
2536                             drv_sectohz(1));
2536                             drv_usectohz(1000000));
2537                         break;
2538                 case IEEE80211_M_MONITOR:
2539                         break;
2540                 }
2541                 rtw_set_nettype(rsc, ic->ic_opmode);
2542                 break;
2543         case IEEE80211_S_ASSOC:
2544         case IEEE80211_S_AUTH:
2545                 break;
2546         }

2548         mutex_exit(&rsc->sc_genlock);
2549         /*
2550          * Invoke the parent method to complete the work.
2551          */
2552         error = rsc->sc_newstate(ic, nstate, arg);

2554         return (error);
2555 }
______unchanged_portion_omitted_
```

```
**********************************************************
  321667 Wed Aug 19 07:25:11 2015
new/usr/src/uts/common/io/sata/adapters/ahci/ahci.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 354 /* The following variables are watchdog handler related */
 355 static clock_t ahci_watchdog_timeout = 5; /* 5 seconds */
 356 static clock_t ahci_watchdog_tick;

 358 /*
 359  * This static variable indicates the size of command table,
 360  * and it's changeable with prdt number, which ahci_dma_prdt_number
 361  * indicates.
 362  */
 363 static size_t ahci_cmd_table_size;

 365 /*
 366  * The below global variables are tunable via /etc/system
 367  *
 368  *      ahci_dma_prdt_number
 369  *      ahci_msi_enabled
 370  *      ahci_buf_64bit_dma
 371  *      ahci_commu_64bit_dma
 372  */

 374 /* The number of Physical Region Descriptor Table(PRDT) in Command Table */
 375 int ahci_dma_prdt_number = AHCI_PRDT_NUMBER;

 377 /* AHCI MSI is tunable */
 378 boolean_t ahci_msi_enabled = B_TRUE;

 380 /*
 381  * 64-bit dma addressing for data buffer is tunable
 382  *
 383  * The variable controls only the below value:
 384  *      DBAU (upper 32-bits physical address of data block)
 385  */
 386 boolean_t ahci_buf_64bit_dma = B_TRUE;

 388 /*
 389  * 64-bit dma addressing for communication system descriptors is tunable
 390  *
 391  * The variable controls the below three values:
 392  *
 393  *      PxCLBU (upper 32-bits for the command list base physical address)
 394  *      PxFBU (upper 32-bits for the received FIS base physical address)
 395  *      CTBAU (upper 32-bits of command table base)
 396  */
 397 boolean_t ahci_commu_64bit_dma = B_TRUE;

 399 /*
 400  * By default, 64-bit dma for data buffer will be disabled for AMD/ATI SB600
 401  * chipset. If the users want to have a try with 64-bit dma, please change
 402  * the below variable value to enable it.
 403  */
 404 boolean_t sb600_buf_64bit_dma_disable = B_TRUE;

 406 /*
 407  * By default, 64-bit dma for command buffer will be disabled for AMD/ATI
 408  * SB600/700/710/750/800. If the users want to have a try with 64-bit dma,
 409  * please change the below value to enable it.
 410  */
 411 boolean_t sbxxx_commu_64bit_dma_disable = B_TRUE;
```



```
 414 /*
 415  * End of global tunable variable definition
 416  */

 418 #if AHCI_DEBUG
 419 uint32_t ahci_debug_flags = 0;
 420 #else
 421 uint32_t ahci_debug_flags = (AHCIDBG_ERRS|AHCIDBG_TIMEOUT);
 422 #endif


 425 #if AHCI_DEBUG
 426 /* The following is needed for ahci_log() */
 427 static kmutex_t ahci_log_mutex;
 428 static char ahci_log_buf[512];
 429 #endif

 431 /* Opaque state pointer initialized by ddi_soft_state_init() */
 432 static void *ahci_statep = NULL;

 434 /*
 435  *  ahci module initialization.
 436  */
 437 int
 438 _init(void)
 439 {
 440         int     ret;

 442         ret = ddi_soft_state_init(&ahci_statep, sizeof (ahci_ctl_t), 0);
 443         if (ret != 0) {
 444                 goto err_out;
 445         }

 447 #if AHCI_DEBUG
 448         mutex_init(&ahci_log_mutex, NULL, MUTEX_DRIVER, NULL);
 449 #endif

 451         if ((ret = sata_hba_init(&modlinkage)) != 0) {
 452 #if AHCI_DEBUG
 453                 mutex_destroy(&ahci_log_mutex);
 454 #endif
 455                 ddi_soft_state_fini(&ahci_statep);
 456                 goto err_out;
 457         }

 459         /* watchdog tick */
 460         ahci_watchdog_tick = drv_sectohz(ahci_watchdog_timeout);
 460         ahci_watchdog_tick = drv_usectohz(
 461             (clock_t)ahci_watchdog_timeout * 1000000);

 462         ret = mod_install(&modlinkage);
 463         if (ret != 0) {
 464                 sata_hba_fini(&modlinkage);
 465 #if AHCI_DEBUG
 466                 mutex_destroy(&ahci_log_mutex);
 467 #endif
 468                 ddi_soft_state_fini(&ahci_statep);
 469                 goto err_out;
 470         }

 472         return (ret);

 474 err_out:
 475         cmn_err(CE_WARN, "!ahci: Module init failed");
 476         return (ret);
```

```
 477 }
_____unchanged_portion_omitted_

1784 /*
1785  * SATA_OPMODE_SYNCH flag is set
1786  *
1787  * If SATA_OPMODE_POLLING flag is set, then we must poll the command
1788  * without interrupt, otherwise we can still use the interrupt.
1789  */
1790 static int
1791 ahci_do_sync_start(ahci_ctl_t *ahci_ctlp, ahci_port_t *ahci_portp,
1792     ahci_addr_t *addrp, sata_pkt_t *spkt)
1793 {
1794         int pkt_timeout_ticks;
1795         uint32_t timeout_tags;
1796         int rval;
1797         int instance = ddi_get_instance(ahci_ctlp->ahcictl_dip);
1798         uint8_t port = addrp->aa_port;

1800         ASSERT(MUTEX_HELD(&ahci_portp->ahciport_mutex));

1802         AHCIDBG(AHCIDBG_ENTRY, ahci_ctlp, "ahci_do_sync_start enter: "
1803             "port %d:%d spkt 0x%p", port, addrp->aa_pmport, spkt);

1805         if (spkt->satapkt_op_mode & SATA_OPMODE_POLLING) {
1806                 ahci_portp->ahciport_flags |= AHCI_PORT_FLAG_POLLING;
1807                 if ((rval = ahci_deliver_satapkt(ahci_ctlp, ahci_portp,
1808                     addrp, spkt)) == AHCI_FAILURE) {
1809                         ahci_portp->ahciport_flags &= ~AHCI_PORT_FLAG_POLLING;
1810                         return (rval);
1811                 }

1813                 pkt_timeout_ticks =
1814                     drv_sectohz((clock_t)spkt->satapkt_time);
1815                     drv_usectohz((clock_t)spkt->satapkt_time * 1000000);

1816                 while (spkt->satapkt_reason == SATA_PKT_BUSY) {
1817                         mutex_exit(&ahci_portp->ahciport_mutex);

1819                         /* Simulate the interrupt */
1820                         ahci_port_intr(ahci_ctlp, ahci_portp, port);

1822                         drv_usecwait(AHCI_10MS_USECS);

1824                         mutex_enter(&ahci_portp->ahciport_mutex);
1825                         pkt_timeout_ticks -= AHCI_10MS_TICKS;
1826                         if (pkt_timeout_ticks < 0) {
1827                                 cmn_err(CE_WARN, "!ahci%d: ahci_do_sync_start "
1828                                     "port %d satapkt 0x%p timed out\n",
1829                                     instance, port, (void *)spkt);
1830                                 timeout_tags = (0x1 << rval);
1831                                 mutex_exit(&ahci_portp->ahciport_mutex);
1832                                 ahci_timeout_pkts(ahci_ctlp, ahci_portp,
1833                                     port, timeout_tags);
1834                                 mutex_enter(&ahci_portp->ahciport_mutex);
1835                         }
1836                 }
1837                 ahci_portp->ahciport_flags &= ~AHCI_PORT_FLAG_POLLING;
1838                 return (AHCI_SUCCESS);

1840         } else {
1841                 if ((rval = ahci_deliver_satapkt(ahci_ctlp, ahci_portp,
1842                     addrp, spkt)) == AHCI_FAILURE)
1843                         return (rval);

1845 #if AHCI_DEBUG
```



```
1846                 /*
1847                  * Note that the driver always uses the slot 0 to deliver
1848                  * REQUEST SENSE or READ LOG EXT command
1849                  */
1850                 if (ERR_RETRI_CMD_IN_PROGRESS(ahci_portp))
1851                         ASSERT(rval == 0);
1852 #endif

1854                 while (spkt->satapkt_reason == SATA_PKT_BUSY)
1855                         cv_wait(&ahci_portp->ahciport_cv,
1856                             &ahci_portp->ahciport_mutex);

1858                 return (AHCI_SUCCESS);
1859         }
1860 }
_____unchanged_portion_omitted_
```

```
**********************************************************
  180627 Wed Aug 19 07:25:11 2015
new/usr/src/uts/common/io/sata/adapters/nv_sata/nv_sata.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


5608 /*
5609  * timeout processing:
5610  *
5611  * Check if any packets have crossed a timeout threshold.  If so,
5612  * abort the packet.  This function is not NCQ-aware.
5613  *
5614  * If reset is in progress, call reset monitoring function.
5615  *
5616  * Timeout frequency may be lower for checking packet timeout
5617  * and higher for reset monitoring.
5618  *
5619  */
5620 static void
5621 nv_timeout(void *arg)
5622 {
5623         nv_port_t *nvp = arg;
5624         nv_slot_t *nv_slotp;
5625         clock_t next_timeout_us = NV_ONE_SEC;
5626         uint16_t int_status;
5627         uint8_t status, bmstatus;
5628         static int intr_warn_once = 0;
5629         uint32_t serror;


5632         ASSERT(nvp != NULL);

5634         mutex_enter(&nvp->nvp_mutex);
5635         nvp->nvp_timeout_id = 0;

5637         if (nvp->nvp_state & (NV_DEACTIVATED|NV_FAILED)) {
5638                 next_timeout_us = 0;

5640                 goto finished;
5641         }

5643         if (nvp->nvp_state & NV_RESET) {
5644                 next_timeout_us = nv_monitor_reset(nvp);

5646                 goto finished;
5647         }

5649         if (nvp->nvp_state & NV_LINK_EVENT) {
5650                 boolean_t device_present = B_FALSE;
5651                 uint32_t sstatus;
5652                 ddi_acc_handle_t bar5_hdl = nvp->nvp_ctlp->nvc_bar_hdl[5];

5654                 if (TICK_TO_USEC(ddi_get_lbolt() -
5655                     nvp->nvp_link_event_time) < NV_LINK_EVENT_SETTLE) {

5657                         next_timeout_us = 10 * NV_ONE_MSEC;

5659                         DTRACE_PROBE(link_event_set_no_timeout_keep_waiting_p);

5661                         goto finished;
5662                 }

5664                 DTRACE_PROBE(link_event_settled_now_process_p);
```



```
5666                 nvp->nvp_state &= ~NV_LINK_EVENT;

5668                 /*
5669                  * ck804 routinely reports the wrong hotplug/unplug event,
5670                  * and it's been seen on mcp55 when there are signal integrity
5671                  * issues.  Therefore need to infer the event from the
5672                  * current link status.
5673                  */

5675                 sstatus = nv_get32(bar5_hdl, nvp->nvp_sstatus);

5677                 if ((SSTATUS_GET_IPM(sstatus) == SSTATUS_IPM_ACTIVE) &&
5678                     (SSTATUS_GET_DET(sstatus) ==
5679                     SSTATUS_DET_DEVPRE_PHYCOM)) {
5680                         device_present = B_TRUE;
5681                 }

5683                 if ((nvp->nvp_signature != NV_NO_SIG) &&
5684                     (device_present == B_FALSE)) {

5686                         NVLOG(NVDBG_HOT, nvp->nvp_ctlp, nvp,
5687                             "nv_timeout: device detached", NULL);

5689                         DTRACE_PROBE(device_detached_p);

5691                         (void) nv_abort_active(nvp, NULL, SATA_PKT_PORT_ERROR,
5692                             B_FALSE);

5694                         nv_port_state_change(nvp, SATA_EVNT_DEVICE_DETACHED,
5695                             SATA_ADDR_CPORT, 0);

5697                         nvp->nvp_signature = NV_NO_SIG;
5698                         nvp->nvp_rem_time = ddi_get_lbolt();
5699                         nvp->nvp_type = SATA_DTYPE_NONE;
5700                         next_timeout_us = 0;

5702 #ifdef SGPIO_SUPPORT
5703                         nv_sgp_drive_disconnect(nvp->nvp_ctlp,
5704                             SGP_CTLR_PORT_TO_DRV(nvp->nvp_ctlp->nvc_ctlr_num,
5705                             nvp->nvp_port_num));
5706 #endif

5708                         goto finished;
5709                 }

5711                 /*
5712                  * if the device was already present, and it's still present,
5713                  * then abort any outstanding command and issue a reset.
5714                  * This may result from transient link errors.
5715                  */

5717                 if ((nvp->nvp_signature != NV_NO_SIG) &&
5718                     (device_present == B_TRUE)) {

5720                         NVLOG(NVDBG_HOT, nvp->nvp_ctlp, nvp,
5721                             "nv_timeout: spurious link event", NULL);
5722                         DTRACE_PROBE(spurious_link_event_p);

5724                         (void) nv_abort_active(nvp, NULL, SATA_PKT_PORT_ERROR,
5725                             B_FALSE);

5727                         nvp->nvp_signature = NV_NO_SIG;
5728                         nvp->nvp_trans_link_time = ddi_get_lbolt();
5729                         nvp->nvp_trans_link_count++;
5730                         next_timeout_us = 0;
```

```
5732                        nv_reset(nvp, "transient link event");

5734                        goto finished;
5735                }


5738                /*
5739                 * a new device has been inserted
5740                 */
5741                if ((nvp->nvp_signature == NV_NO_SIG) &&
5742                    (device_present == B_TRUE)) {
5743                        NVLOG(NVDBG_HOT, nvp->nvp_ctlp, nvp,
5744                            "nv_timeout: device attached", NULL);

5746                        DTRACE_PROBE(device_attached_p);
5747                        nvp->nvp_add_time = ddi_get_lbolt();
5748                        next_timeout_us = 0;
5749                        nvp->nvp_reset_count = 0;
5750                        nvp->nvp_state = NV_HOTPLUG;
5751                        nvp->nvp_type = SATA_DTYPE_UNKNOWN;
5752                        nv_reset(nvp, "hotplug");

5754                        goto finished;
5755                }

5757                /*
5758                 * no link, and no prior device.  Nothing to do, but
5759                 * log this.
5760                 */
5761                NVLOG(NVDBG_HOT, nvp->nvp_ctlp, nvp,
5762                    "nv_timeout: delayed hot processing no link no prior"
5763                    " device", NULL);
5764                DTRACE_PROBE(delayed_hotplug_no_link_no_prior_device_p);

5766                nvp->nvp_trans_link_time = ddi_get_lbolt();
5767                nvp->nvp_trans_link_count++;
5768                next_timeout_us = 0;

5770                goto finished;
5771        }

5773        /*
5774         * Not yet NCQ-aware - there is only one command active.
5775         */
5776        nv_slotp = &(nvp->nvp_slot[0]);

5778        /*
5779         * perform timeout checking and processing only if there is an
5780         * active packet on the port
5781         */
5782        if (nv_slotp != NULL && nv_slotp->nvslot_spkt != NULL)  {
5783                sata_pkt_t *spkt = nv_slotp->nvslot_spkt;
5784                sata_cmd_t *satacmd = &spkt->satapkt_cmd;
5785                uint8_t cmd = satacmd->satacmd_cmd_reg;
5786                uint64_t lba;

5788 #if ! defined(__lock_lint) && defined(DEBUG)

5790                lba = (uint64_t)satacmd->satacmd_lba_low_lsb |
5791                    ((uint64_t)satacmd->satacmd_lba_mid_lsb << 8) |
5792                    ((uint64_t)satacmd->satacmd_lba_high_lsb << 16) |
5793                    ((uint64_t)satacmd->satacmd_lba_low_msb << 24) |
5794                    ((uint64_t)satacmd->satacmd_lba_mid_msb << 32) |
5795                    ((uint64_t)satacmd->satacmd_lba_high_msb << 40);
5796 #endif
```



```
5798                /*
5799                 * timeout not needed if there is a polling thread
5800                 */
5801                if (spkt->satapkt_op_mode & SATA_OPMODE_POLLING) {
5802                        next_timeout_us = 0;

5804                        goto finished;
5805                }

5807                if (TICK_TO_SEC(ddi_get_lbolt() - nv_slotp->nvslot_stime) >
5808                    spkt->satapkt_time) {

5810                        serror = nv_get32(nvp->nvp_ctlp->nvc_bar_hdl[5],
5811                            nvp->nvp_serror);
5812                        status = nv_get8(nvp->nvp_ctl_hdl,
5813                            nvp->nvp_altstatus);
5814                        bmstatus = nv_get8(nvp->nvp_bm_hdl,
5815                            nvp->nvp_bmisx);

5817                        nv_cmn_err(CE_NOTE, nvp->nvp_ctlp, nvp,
5818                            "nv_timeout: aborting: "
5819                            "nvslot_stime: %ld max ticks till timeout: %ld "
5820                            "cur_time: %ld cmd = 0x%x lba = %d seq = %d",
5821                            nv_slotp->nvslot_stime,
5822                            drv_sectohz(spkt->satapkt_time), ddi_get_lbolt(),
5822                            drv_usectohz(MICROSEC *
5823                            spkt->satapkt_time), ddi_get_lbolt(),
5823                            cmd, lba, nvp->nvp_seq);

5825                        NVLOG(NVDBG_TIMEOUT, nvp->nvp_ctlp, nvp,
5826                            "nv_timeout: altstatus = 0x%x  bmicx = 0x%x "
5827                            "serror = 0x%x previous_cmd = "
5828                            "0x%x", status, bmstatus, serror,
5829                            nvp->nvp_previous_cmd);


5832                        DTRACE_PROBE1(nv_timeout_packet_p, int, nvp);

5834                        if (nvp->nvp_mcp5x_int_status != NULL) {

5836                                int_status = nv_get16(
5837                                    nvp->nvp_ctlp->nvc_bar_hdl[5],
5838                                    nvp->nvp_mcp5x_int_status);
5839                                NVLOG(NVDBG_TIMEOUT, nvp->nvp_ctlp, nvp,
5840                                    "int_status = 0x%x", int_status);

5842                                if (int_status & MCP5X_INT_COMPLETE) {
5843                                        /*
5844                                         * Completion interrupt was missed.
5845                                         * Issue warning message once.
5846                                         */
5847                                        if (!intr_warn_once) {

5849                                                nv_cmn_err(CE_WARN,
5850                                                    nvp->nvp_ctlp,
5851                                                    nvp,
5852                                                    "nv_sata: missing command "
5853                                                    "completion interrupt");
5854                                                intr_warn_once = 1;

5856                                        }

5858                                        NVLOG(NVDBG_TIMEOUT, nvp->nvp_ctlp,
5859                                            nvp, "timeout detected with "
5860                                            "interrupt ready - calling "
5861                                            "int directly", NULL);
```

```
5863                                        mutex_exit(&nvp->nvp_mutex);
5864                                        (void) mcp5x_intr_port(nvp);
5865                                        mutex_enter(&nvp->nvp_mutex);

5867                                } else {
5868                                        /*
5869                                         * True timeout and not a missing
5870                                         * interrupt.
5871                                         */
5872                                        DTRACE_PROBE1(timeout_abort_active_p,
5873                                            int *, nvp);
5874                                        (void) nv_abort_active(nvp, spkt,
5875                                            SATA_PKT_TIMEOUT, B_TRUE);
5876                                }
5877                        } else {
5878                                (void) nv_abort_active(nvp, spkt,
5879                                    SATA_PKT_TIMEOUT, B_TRUE);
5880                        }

5882                } else {
5883                        NVLOG(NVDBG_VERBOSE, nvp->nvp_ctlp, nvp,
5884                            "nv_timeout:"
5885                            " still in use so restarting timeout",
5886                            NULL);

5888                        next_timeout_us = NV_ONE_SEC;
5889                }
5890        } else {
5891                /*
5892                 * there was no active packet, so do not re-enable timeout
5893                 */
5894                next_timeout_us = 0;
5895                NVLOG(NVDBG_VERBOSE, nvp->nvp_ctlp, nvp,
5896                    "nv_timeout: no active packet so not re-arming "
5897                    "timeout", NULL);
5898        }

5900 finished:

5902        nv_setup_timeout(nvp, next_timeout_us);

5904        mutex_exit(&nvp->nvp_mutex);
5905 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  165038 Wed Aug 19 07:25:11 2015
new/usr/src/uts/common/io/sata/adapters/si3124/si3124.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 424 /* The following are needed for si_log() */
 425 static kmutex_t si_log_mutex;
 426 static char si_log_buf[SI_LOGBUF_LEN];
 427 uint32_t si_debug_flags =
 428     SIDBG_ERRS|SIDBG_INIT|SIDBG_EVENT|SIDBG_TIMEOUT|SIDBG_RESET;

 430 static int is_msi_supported = 0;

 432 /*
 433  * The below global variables are tunable via /etc/system
 434  *
 435  * si_dma_sg_number
 436  */

 438 int si_dma_sg_number = SI_DEFAULT_SGT_TABLES_PER_PRB;

 440 /* Opaque state pointer to be initialized by ddi_soft_state_init() */
 441 static void *si_statep  = NULL;

 443 /*
 444  *  si3124 module initialization.
 445  *
 446  */
 447 int
 448 _init(void)
 449 {
 450         int     error;

 452         error = ddi_soft_state_init(&si_statep, sizeof (si_ctl_state_t), 0);
 453         if (error != 0) {
 454                 return (error);
 455         }

 457         mutex_init(&si_log_mutex, NULL, MUTEX_DRIVER, NULL);

 459         if ((error = sata_hba_init(&modlinkage)) != 0) {
 460                 mutex_destroy(&si_log_mutex);
 461                 ddi_soft_state_fini(&si_statep);
 462                 return (error);
 463         }

 465         error = mod_install(&modlinkage);
 466         if (error != 0) {
 467                 sata_hba_fini(&modlinkage);
 468                 mutex_destroy(&si_log_mutex);
 469                 ddi_soft_state_fini(&si_statep);
 470                 return (error);
 471         }

 473         si_watchdog_tick = drv_sectohz((clock_t)si_watchdog_timeout);
 473         si_watchdog_tick = drv_usectohz((clock_t)si_watchdog_timeout * 1000000);

 475         return (error);
 476 }
______unchanged_portion_omitted_


2327 /*
```



```
2328  * Polls for the completion of the command. This is safe with both
2329  * interrupts enabled or disabled.
2330  */
2331 static void
2332 si_poll_cmd(
2333         si_ctl_state_t *si_ctlp,
2334         si_port_state_t *si_portp,
2335         int port,
2336         int slot,
2337         sata_pkt_t *satapkt)
2338 {
2339         uint32_t slot_status;
2340         int pkt_timeout_ticks;
2341         uint32_t port_intr_status;
2342         int in_panic = ddi_in_panic();

2344         SIDBG_P(SIDBG_ENTRY, si_portp, "si_poll_cmd entered: port: 0x%x", port);

2346         pkt_timeout_ticks = drv_sectohz((clock_t)satapkt->satapkt_time);
2346         pkt_timeout_ticks = drv_usectohz((clock_t)satapkt->satapkt_time *
2347             1000000);


2349         /* we start out with SATA_PKT_COMPLETED as the satapkt_reason */
2350         satapkt->satapkt_reason = SATA_PKT_COMPLETED;

2352         do {
2353                 slot_status = ddi_get32(si_ctlp->sictl_port_acc_handle,
2354                     (uint32_t *)(PORT_SLOT_STATUS(si_ctlp, port)));

2356                 if (slot_status & SI_SLOT_MASK & (0x1 << slot)) {
2357                         if (in_panic) {
2358                                 /*
2359                                  * If we are in panic, we can't rely on
2360                                  * timers; so, busy wait instead of delay().
2361                                  */
2362                                 mutex_exit(&si_portp->siport_mutex);
2363                                 drv_usecwait(SI_1MS_USECS);
2364                                 mutex_enter(&si_portp->siport_mutex);
2365                         } else {
2366                                 mutex_exit(&si_portp->siport_mutex);
2367 #ifndef __lock_lint
2368                                 delay(SI_1MS_TICKS);
2369 #endif /* __lock_lint */
2370                                 mutex_enter(&si_portp->siport_mutex);
2371                         }
2372                 } else {
2373                         break;
2374                 }

2376                 pkt_timeout_ticks -= SI_1MS_TICKS;

2378         } while (pkt_timeout_ticks > 0);

2380         if (satapkt->satapkt_reason != SATA_PKT_COMPLETED) {
2381                 /* The si_mop_command() got to our packet before us */

2383                 return;
2384         }

2386         /*
2387          * Interrupts and timers may not be working properly in a crash dump
2388          * situation; we may need to handle all the three conditions here:
2389          * successful completion, packet failure and packet timeout.
2390          */
2391         if (IS_ATTENTION_RAISED(slot_status)) { /* error seen on port */
```

```
2393                port_intr_status = ddi_get32(si_ctlp->sictl_global_acc_handle,
2394                    (uint32_t *)PORT_INTERRUPT_STATUS(si_ctlp, port));

2396                SIDBG_P(SIDBG_VERBOSE, si_portp,
2397                    "si_poll_cmd: port_intr_status: 0x%x, port: %x",
2398                    port_intr_status, port);

2400                if (port_intr_status & INTR_COMMAND_ERROR) {
2401                        mutex_exit(&si_portp->siport_mutex);
2402                        (void) si_intr_command_error(si_ctlp, si_portp, port);
2403                        mutex_enter(&si_portp->siport_mutex);

2405                        return;

2407                        /*
2408                         * Why do we need to call si_intr_command_error() ?
2409                         *
2410                         * Answer: Even if the current packet is not the
2411                         * offending command, we need to restart the stalled
2412                         * port; (may be, the interrupts are not working well
2413                         * in panic condition). The call to routine
2414                         * si_intr_command_error() will achieve that.
2415                         *
2416                         * What if the interrupts are working fine and the
2417                         * si_intr_command_error() gets called once more from
2418                         * interrupt context ?
2419                         *
2420                         * Answer: The second instance of routine
2421                         * si_intr_command_error() will not mop anything
2422                         * since the first error handler has already blown
2423                         * away the hardware pending queues through reset.
2424                         *
2425                         * Will the si_intr_command_error() hurt current
2426                         * packet ?
2427                         *
2428                         * Answer: No.
2429                         */
2430                } else {
2431                        /* Ignore any non-error interrupts at this stage */
2432                        ddi_put32(si_ctlp->sictl_port_acc_handle,
2433                            (uint32_t *)(PORT_INTERRUPT_STATUS(si_ctlp,
2434                            port)),
2435                            port_intr_status & INTR_MASK);
2436                }

2438        } else if (slot_status & SI_SLOT_MASK & (0x1 << slot)) {
2439                satapkt->satapkt_reason = SATA_PKT_TIMEOUT;

2441        } /* else: the command completed successfully */

2443        if (satapkt->satapkt_cmd.satacmd_flags.sata_special_regs) {
2444                si_copy_out_regs(&satapkt->satapkt_cmd, si_ctlp, port, slot);
2445        }

2447        if ((satapkt->satapkt_cmd.satacmd_cmd_reg ==
2448            SATAC_WRITE_FPDMA_QUEUED) ||
2449            (satapkt->satapkt_cmd.satacmd_cmd_reg ==
2450            SATAC_READ_FPDMA_QUEUED)) {
2451                si_portp->siport_pending_ncq_count--;
2452        }

2454        CLEAR_BIT(si_portp->siport_pending_tags, slot);

2456        /*
2457         * tidbit: What is the interaction of abort with polling ?
```



```
2458         * What happens if the current polled pkt is aborted in parallel ?
2459         *
2460         * Answer: Assuming that the si_mop_commands() completes ahead
2461         * of polling, all it does is to set the satapkt_reason to
2462         * SPKT_PKT_ABORTED. That would be fine with us.
2463         *
2464         * The same logic applies to reset interacting with polling.
2465         */
2466 }
```

**_____*unchanged_portion_omitted_*

```
**********************************************************
  637166 Wed Aug 19 07:25:12 2015
new/usr/src/uts/common/io/sata/impl/sata.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

6748 /*
6749  * Re-identify device after doing a firmware download.
6750  */
6751 static void
6752 sata_reidentify_device(sata_pkt_txlate_t *spx)
6753 {
6754 #define DOWNLOAD_WAIT_TIME_SECS 60
6755 #define DOWNLOAD_WAIT_INTERVAL_SECS     1
6756         int rval;
6757         int retry_cnt;
6758         struct scsi_pkt *scsipkt = spx->txlt_scsi_pkt;
6759         sata_hba_inst_t *sata_hba_inst = spx->txlt_sata_hba_inst;
6760         sata_device_t sata_device = spx->txlt_sata_pkt->satapkt_device;
6761         sata_drive_info_t *sdinfo;

6763         /*
6764          * Before returning good status, probe device.
6765          * Device probing will get IDENTIFY DEVICE data, if possible.
6766          * The assumption is that the new microcode is applied by the
6767          * device. It is a caller responsibility to verify this.
6768          */
6769         for (retry_cnt = 0;
6770             retry_cnt < DOWNLOAD_WAIT_TIME_SECS / DOWNLOAD_WAIT_INTERVAL_SECS;
6771             retry_cnt++) {
6772                 rval = sata_probe_device(sata_hba_inst, &sata_device);

6774                 if (rval == SATA_SUCCESS) { /* Set default features */
6775                         sdinfo = sata_get_device_info(sata_hba_inst,
6776                             &sata_device);
6777                         if (sata_initialize_device(sata_hba_inst, sdinfo) !=
6778                             SATA_SUCCESS) {
6779                                 /* retry */
6780                                 rval = sata_initialize_device(sata_hba_inst,
6781                                     sdinfo);
6782                                 if (rval == SATA_RETRY)
6783                                         sata_log(sata_hba_inst, CE_WARN,
6784                                             "SATA device at port %d pmport %d -"
6785                                             " default device features could not"
6786                                             " be set. Device may not operate "
6787                                             "as expected.",
6788                                             sata_device.satadev_addr.cport,
6789                                             sata_device.satadev_addr.pmport);
6790                         }
6791                         if ((scsipkt->pkt_flags & FLAG_NOINTR) == 0)
6792                                 scsi_hba_pkt_comp(scsipkt);
6793                         return;
6794                 } else if (rval == SATA_RETRY) {
6795                         delay(drv_sectohz(DOWNLOAD_WAIT_INTERVAL_SECS));
6795                         delay(drv_usectohz(1000000 *
6796                             DOWNLOAD_WAIT_INTERVAL_SECS));
6796                         continue;
6797                 } else  /* failed - no reason to retry */
6798                         break;
6799         }

6801         /*
6802          * Something went wrong, device probing failed.
6803          */
6804         SATA_LOG_D((sata_hba_inst, CE_WARN,
```



```
6805             "Cannot probe device after downloading microcode\n"));

6807         /* Reset device to force retrying the probe. */
6808         (void) (*SATA_RESET_DPORT_FUNC(sata_hba_inst))
6809             (SATA_DIP(sata_hba_inst), &sata_device);

6811         if ((scsipkt->pkt_flags & FLAG_NOINTR) == 0)
6812                 scsi_hba_pkt_comp(scsipkt);
6813 }
______unchanged_portion_omitted_
```

```
**********************************************************
   42384 Wed Aug 19 07:25:12 2015
new/usr/src/uts/common/io/sbp2/sbp2.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 864 /*
 865  * This is where tasks (command ORB's) are signalled to the target.
 866  * 'task' argument is allowed to be NULL, in which case the task will be
 867  * taken from the current task list.
 868  *
 869  * Tasks are signalled one at a time by writing into ORB_POINTER register.
 870  * While SBP-2 allows dynamic task list updates and using DOORBELL register,
 871  * some devices have bugs that prevent using this strategy: e.g. some LaCie
 872  * HDD's can corrupt data. Data integrity is more important than performance.
 873  */
 874 int
 875 sbp2_ses_submit_task(sbp2_ses_t *sp, sbp2_task_t *new_task)
 876 {
 877         sbp2_agent_t    *ap = &sp->s_agent;
 878         sbp2_tgt_t      *tp = sp->s_tgt;
 879         sbp2_task_t     *task;          /* task actually being submitted */
 880         boolean_t       callback;
 881         timeout_id_t    timeout_id;
 882         int             ret;

 884         if (!sbp2_lun_accepting_tasks(sp->s_lun)) {
 885                 return (SBP2_ENODEV);
 886         }

 888         sbp2_agent_acquire(ap); /* serialize */

 890         mutex_enter(&ap->a_mutex);

 892         /* if task provided, append it to the list */
 893         if (new_task != NULL) {
 894                 ASSERT(new_task->ts_state == SBP2_TASK_INIT);
 895                 sbp2_ses_append_task(sp, new_task);
 896         }

 898         /* if there is already a task in flight, exit */
 899         if ((ap->a_active_task != NULL) &&
 900             (ap->a_active_task->ts_state == SBP2_TASK_PEND)) {
 901                 mutex_exit(&ap->a_mutex);
 902                 sbp2_agent_release(ap);
 903                 return (SBP2_SUCCESS);
 904         }

 906         /*
 907          * cannot submit tasks from interrupt context,
 908          * upper layer driver is responsible to call nudge
 909          */
 910         if (servicing_interrupt()) {
 911                 mutex_exit(&ap->a_mutex);
 912                 sbp2_agent_release(ap);
 913                 return (SBP2_ECONTEXT);
 914         }

 916         /* no active task, grab the first one on the list in INIT state */
 917         ap->a_active_task = sbp2_ses_find_task_state(sp, SBP2_TASK_INIT);
 918         if (ap->a_active_task == NULL) {
 919                 mutex_exit(&ap->a_mutex);
 920                 sbp2_agent_release(ap);
 921                 return (SBP2_SUCCESS);
 922         }
```



```
 923         task = ap->a_active_task;
 924         task->ts_ses = sp;
 925         task->ts_state = SBP2_TASK_PEND;

 927         /* can't work with a dead agent */
 928         if (sbp2_agent_keepalive(ap, &task->ts_bus_error) != SBP2_SUCCESS) {
 929                 task->ts_error = SBP2_TASK_ERR_DEAD;
 930                 goto error;
 931         }

 933         /*
 934          * In theory, we should schedule task timeout after it's been submitted.
 935          * However, some fast tasks complete even before timeout is scheduled.
 936          * To avoid additional complications in the code, schedule timeout now.
 937          */
 938         ASSERT(task->ts_timeout_id == 0);
 939         task->ts_time_start = gethrtime();
 940         if (task->ts_timeout > 0) {
 941                 task->ts_timeout_id = timeout(sbp2_task_timeout, task,
 942                     drv_sectohz(task->ts_timeout));
 942                     task->ts_timeout * drv_usectohz(1000000));
 943         }

 945         /* notify fetch agent */
 946         ap->a_state = SBP2_AGENT_STATE_ACTIVE;
 947         mutex_exit(&ap->a_mutex);
 948         ret = sbp2_agent_write_orbp(ap, task->ts_buf->bb_baddr,
 949             &task->ts_bus_error);
 950         tp->t_stat.stat_submit_orbp++;
 951         mutex_enter(&ap->a_mutex);

 953         if (ret != SBP2_SUCCESS) {
 954                 ap->a_state = SBP2_AGENT_STATE_DEAD;
 955                 tp->t_stat.stat_status_dead++;

 957                 if (task->ts_timeout_id != 0) {
 958                         timeout_id = task->ts_timeout_id;
 959                         task->ts_timeout_id = 0;
 960                         (void) untimeout(timeout_id);
 961                 }
 962                 task->ts_error = SBP2_TASK_ERR_BUS;
 963                 goto error;
 964         }

 966         mutex_exit(&ap->a_mutex);

 968         sbp2_agent_release(ap);
 969         return (SBP2_SUCCESS);

 971 error:
 972         /*
 973          * Return immediate error if failed task is the one being submitted,
 974          * otherwise use callback.
 975          */
 976         callback = (ap->a_active_task != new_task);
 977         ASSERT(task == ap->a_active_task);
 978         ap->a_active_task = NULL;
 979         mutex_exit(&ap->a_mutex);
 980         sbp2_agent_release(ap);

 982         /*
 983          * Remove task from the list. It is important not to change task state
 984          * to SBP2_TASK_COMP while it's still on the list, to avoid race with
 985          * upper layer driver (e.g. scsa1394).
 986          */
 987         ret = sbp2_ses_remove_task(sp, task);
```

```
 988         ASSERT(ret == SBP2_SUCCESS);
 989         task->ts_state = SBP2_TASK_COMP;

 991         if (callback) {
 992                 sp->s_status_cb(sp->s_status_cb_arg, task);
 993                 return (SBP2_SUCCESS);
 994         } else {
 995                 /* upper layer driver is responsible to call nudge */
 996                 return (SBP2_FAILURE);
 997         }
 998 }
```

unchanged_portion_omitted_

```
**********************************************************
  449005 Wed Aug 19 07:25:13 2015
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

10044 static int
10045 mptsas_quiesce_bus(mptsas_t *mpt)
10046 {
10047         mptsas_target_t *ptgt = NULL;

10049         NDBG28(("mptsas_quiesce_bus"));
10050         mutex_enter(&mpt->m_mutex);

10052         /* Set all the throttles to zero */
10053         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10054             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10055                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10056         }

10058         /* If there are any outstanding commands in the queue */
10059         if (mpt->m_ncmds) {
10060                 mpt->m_softstate |= MPTSAS_SS_DRAINING;
10061                 mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10062                     mpt, drv_sectohz(MPTSAS_QUIESCE_TIMEOUT));
10062                     mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
10063                 if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
10064                         /*
10065                          * Quiesce has been interrupted
10066                          */
10067                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10068                         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10069                             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10070                                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10071                         }
10072                         mptsas_restart_hba(mpt);
10073                         if (mpt->m_quiesce_timeid != 0) {
10074                                 timeout_id_t tid = mpt->m_quiesce_timeid;
10075                                 mpt->m_quiesce_timeid = 0;
10076                                 mutex_exit(&mpt->m_mutex);
10077                                 (void) untimeout(tid);
10078                                 return (-1);
10079                         }
10080                         mutex_exit(&mpt->m_mutex);
10081                         return (-1);
10082                 } else {
10083                         /* Bus has been quiesced */
10084                         ASSERT(mpt->m_quiesce_timeid == 0);
10085                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10086                         mpt->m_softstate |= MPTSAS_SS_QUIESCED;
10087                         mutex_exit(&mpt->m_mutex);
10088                         return (0);
10089                 }
10090         }
10091         /* Bus was not busy - QUIESCED */
10092         mutex_exit(&mpt->m_mutex);

10094         return (0);
10095 }
______unchanged_portion_omitted_

10114 static void
10115 mptsas_ncmds_checkdrain(void *arg)
10116 {
10117         mptsas_t        *mpt = arg;
```



```
10118         mptsas_target_t *ptgt = NULL;

10120         mutex_enter(&mpt->m_mutex);
10121         if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
10122                 mpt->m_quiesce_timeid = 0;
10123                 if (mpt->m_ncmds == 0) {
10124                         /* Command queue has been drained */
10125                         cv_signal(&mpt->m_cv);
10126                 } else {
10127                         /*
10128                          * The throttle may have been reset because
10129                          * of a SCSI bus reset
10130                          */
10131                         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10132                             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10133                                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10134                         }

10136                         mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10137                             mpt, drv_sectohz(MPTSAS_QUIESCE_TIMEOUT));
10137                             mpt, (MPTSAS_QUIESCE_TIMEOUT *
10138                             drv_usectohz(1000000)));
10138                 }
10139         }
10140         mutex_exit(&mpt->m_mutex);
10141 }
______unchanged_portion_omitted_

13592 uint64_t
13593 mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun)
13594 {
13595         uint64_t        sata_guid = 0, *pwwn = NULL;
13596         int             target = ptgt->m_devhdl;
13597         uchar_t         *inq83 = NULL;
13598         int             inq83_len = 0xFF;
13599         uchar_t         *dblk = NULL;
13600         int             inq83_retry = 3;
13601         int             rval = DDI_FAILURE;

13603         inq83   = kmem_zalloc(inq83_len, KM_SLEEP);

13605 inq83_retry:
13606         rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
13607             inq83_len, NULL, 1);
13608         if (rval != DDI_SUCCESS) {
13609                 mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
13610                     "0x83 for target:%x, lun:%x failed!", target, lun);
13611                 goto out;
13612         }
13613         /* According to SAT2, the first descriptor is logic unit name */
13614         dblk = &inq83[4];
13615         if ((dblk[1] & 0x30) != 0) {
13616                 mptsas_log(mpt, CE_WARN, "!Descriptor is not lun associated.");
13617                 goto out;
13618         }
13619         pwwn = (uint64_t *)(void *)(&dblk[4]);
13620         if ((dblk[4] & 0xf0) == 0x50) {
13621                 sata_guid = BE_64(*pwwn);
13622                 goto out;
13623         } else if (dblk[4] == 'A') {
13624                 NDBG20(("SATA drive has no NAA format GUID."));
13625                 goto out;
13626         } else {
13627                 /* The data is not ready, wait and retry */
13628                 inq83_retry--;
13629                 if (inq83_retry <= 0) {
```

```
13630                           goto out;
13631                   }
13632                   NDBG20(("The GUID is not ready, retry..."));
13633                   delay(drv_sectohz(1));
13634                   delay(1 * drv_usectohz(1000000));
13635                   goto inq83_retry;
13636           }
13637   out:
13638           kmem_free(inq83, inq83_len);
13639           return (sata_guid);
13640   }
```

_____***unchanged_portion_omitted_***

```
14847 static int
14848 mptsas_create_lun(dev_info_t *pdip, struct scsi_inquiry *sd_inq,
14849     dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14850 {
14851           int                     i = 0;
14852           uchar_t                 *inq83 = NULL;
14853           int                     inq83_len1 = 0xFF;
14854           int                     inq83_len = 0;
14855           int                     rval = DDI_FAILURE;
14856           ddi_devid_t             devid;
14857           char                    *guid = NULL;
14858           int                     target = ptgt->m_devhdl;
14859           mdi_pathinfo_t          *pip = NULL;
14860           mptsas_t                *mpt = DIP2MPT(pdip);

14862           /*
14863            * For DVD/CD ROM and tape devices and optical
14864            * devices, we won't try to enumerate them under
14865            * scsi_vhci, so no need to try page83
14866            */
14867           if (sd_inq && (sd_inq->inq_dtype == DTYPE_RODIRECT ||
14868               sd_inq->inq_dtype == DTYPE_OPTICAL ||
14869               sd_inq->inq_dtype == DTYPE_ESI))
14870                   goto create_lun;

14872           /*
14873            * The LCA returns good SCSI status, but corrupt page 83 data the first
14874            * time it is queried. The solution is to keep trying to request page83
14875            * and verify the GUID is not (DDI_NOT_WELL_FORMED) in
14876            * mptsas_inq83_retry_timeout seconds. If the timeout expires, driver
14877            * give up to get VPD page at this stage and fail the enumeration.
14878            */

14880           inq83   = kmem_zalloc(inq83_len1, KM_SLEEP);

14882           for (i = 0; i < mptsas_inq83_retry_timeout; i++) {
14883                   rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
14884                       inq83_len1, &inq83_len, 1);
14885                   if (rval != 0) {
14886                           mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
14887                               "0x83 for target:%x, lun:%x failed!", target, lun);
14888                           if (mptsas_physical_bind_failed_page_83 != B_FALSE)
14889                                   goto create_lun;
14890                           goto out;
14891                   }
14892                   /*
14893                    * create DEVID from inquiry data
14894                    */
14895                   if ((rval = ddi_devid_scsi_encode(
14896                       DEVID_SCSI_ENCODE_VERSION_LATEST, NULL, (uchar_t *)sd_inq,
14897                       sizeof (struct scsi_inquiry), NULL, 0, inq83,
14898                       (size_t)inq83_len, &devid)) == DDI_SUCCESS) {
14899                           /*
```



```
14900                           * extract GUID from DEVID
14901                           */
14902                          guid = ddi_devid_to_guid(devid);

14904                          /*
14905                           * Do not enable MPXIO if the strlen(guid) is greater
14906                           * than MPTSAS_MAX_GUID_LEN, this constrain would be
14907                           * handled by framework later.
14908                           */
14909                          if (guid && (strlen(guid) > MPTSAS_MAX_GUID_LEN)) {
14910                                  ddi_devid_free_guid(guid);
14911                                  guid = NULL;
14912                                  if (mpt->m_mpxio_enable == TRUE) {
14913                                          mptsas_log(mpt, CE_NOTE, "!Target:%x, "
14914                                              "lun:%x doesn't have a valid GUID, "
14915                                              "multipathing for this drive is "
14916                                              "not enabled", target, lun);
14917                                  }
14918                          }

14920                          /*
14921                           * devid no longer needed
14922                           */
14923                          ddi_devid_free(devid);
14924                          break;
14925                  } else if (rval == DDI_NOT_WELL_FORMED) {
14926                          /*
14927                           * return value of ddi_devid_scsi_encode equal to
14928                           * DDI_NOT_WELL_FORMED means DEVID_RETRY, it worth
14929                           * to retry inquiry page 0x83 and get GUID.
14930                           */
14931                          NDBG20(("Not well formed devid, retry..."));
14932                          delay(drv_sectohz(1));
14933                          delay(1 * drv_usectohz(1000000));
14933                          continue;
14934                  } else {
14935                          mptsas_log(mpt, CE_WARN, "!Encode devid failed for "
14936                              "path target:%x, lun:%x", target, lun);
14937                          rval = DDI_FAILURE;
14938                          goto create_lun;
14939                  }
14940          }

14942          if (i == mptsas_inq83_retry_timeout) {
14943                  mptsas_log(mpt, CE_WARN, "!Repeated page83 requests timeout "
14944                      "for path target:%x, lun:%x", target, lun);
14945          }

14947          rval = DDI_FAILURE;

14949 create_lun:
14950          if ((guid != NULL) && (mpt->m_mpxio_enable == TRUE)) {
14951                  rval = mptsas_create_virt_lun(pdip, sd_inq, guid, lun_dip, &pip,
14952                      ptgt, lun);
14953          }
14954          if (rval != DDI_SUCCESS) {
14955                  rval = mptsas_create_phys_lun(pdip, sd_inq, guid, lun_dip,
14956                      ptgt, lun);

14958          }
14959 out:
14960          if (guid != NULL) {
14961                  /*
14962                   * guid no longer needed
14963                   */
14964                  ddi_devid_free_guid(guid);
```

```
14965         }
14966         if (inq83 != NULL)
14967                 kmem_free(inq83, inq83_len1);
14968         return (rval);
14969 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   83465 Wed Aug 19 07:25:13 2015
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1290 static int mptsas_enable_mpi25_flashupdate = 0;

1292 int
1293 mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
1294     uint8_t type, int mode)
1295 {

1297         /*
1298          * In order to avoid allocating variables on the stack,
1299          * we make use of the pre-existing mptsas_cmd_t and
1300          * scsi_pkt which are included in the mptsas_t which
1301          * is passed to this routine.
1302          */

1304         ddi_dma_attr_t          flsh_dma_attrs;
1305         ddi_dma_cookie_t        flsh_cookie;
1306         ddi_dma_handle_t        flsh_dma_handle;
1307         ddi_acc_handle_t        flsh_accessp;
1308         caddr_t                 memp, flsh_memp;
1309         mptsas_cmd_t            *cmd;
1310         struct scsi_pkt         *pkt;
1311         int                     i;
1312         int                     rvalue = 0;
1313         uint64_t                request_desc;

1315         if (mpt->m_MPI25 && !mptsas_enable_mpi25_flashupdate) {
1316                 /*
1317                  * The code is there but not tested yet.
1318                  * User has to know there are risks here.
1319                  */
1320                 mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): "
1321                     "Updating firmware through MPI 2.5 has not been "
1322                     "tested yet!\n"
1323                     "To enable set mptsas_enable_mpi25_flashupdate to 1\n");
1324                 return (-1);
1325         } /* Otherwise, you pay your money and you take your chances. */

1327         if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
1328                 mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): allocation "
1329                     "failed. event ack command pool is full\n");
1330                 return (rvalue);
1331         }

1333         bzero((caddr_t)cmd, sizeof (*cmd));
1334         bzero((caddr_t)pkt, scsi_pkt_size());
1335         cmd->ioc_cmd_slot = (uint32_t)rvalue;

1337         /*
1338          * dynamically create a customized dma attribute structure
1339          * that describes the flash file.
1340          */
1341         flsh_dma_attrs = mpt->m_msg_dma_attr;
1342         flsh_dma_attrs.dma_attr_sgllen = 1;

1344         if (mptsas_dma_addr_create(mpt, flsh_dma_attrs, &flsh_dma_handle,
1345             &flsh_accessp, &flsh_memp, size, &flsh_cookie) == FALSE) {
1346                 mptsas_log(mpt, CE_WARN,
1347                     "(unable to allocate dma resource.");
1348                 mptsas_return_to_pool(mpt, cmd);
```



```
1349                 return (-1);
1350         }

1352         bzero(flsh_memp, size);

1354         for (i = 0; i < size; i++) {
1355                 (void) ddi_copyin(ptrbuffer + i, flsh_memp + i, 1, mode);
1356         }
1357         (void) ddi_dma_sync(flsh_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);

1359         /*
1360          * form a cmd/pkt to store the fw download message
1361          */
1362         pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
1363         pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
1364         pkt->pkt_ha_private      = (opaque_t)cmd;
1365         pkt->pkt_flags          = FLAG_HEAD;
1366         pkt->pkt_time           = 60;
1367         cmd->cmd_pkt            = pkt;
1368         cmd->cmd_scblen         = 1;
1369         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_CMD;

1371         /*
1372          * Save the command in a slot
1373          */
1374         if (mptsas_save_cmd(mpt, cmd) == FALSE) {
1375                 mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);
1376                 mptsas_return_to_pool(mpt, cmd);
1377                 return (-1);
1378         }

1380         /*
1381          * Fill in fw download message
1382          */
1383         ASSERT(cmd->cmd_slot != 0);
1384         memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
1385         bzero(memp, mpt->m_req_frame_size);

1387         if (mpt->m_MPI25)
1388                 mptsas_uflash25((pMpi25FWDownloadRequest)memp,
1389                     mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);
1390         else
1391                 mptsas_uflash2((pMpi2FWDownloadRequest)memp,
1392                     mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);

1394         /*
1395          * Start command
1396          */
1397         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1398             DDI_DMA_SYNC_FORDEV);
1399         request_desc = (cmd->cmd_slot << 16) +
1400             MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
1401         cmd->cmd_rfm = NULL;
1402         MPTSAS_START_CMD(mpt, request_desc);

1404         rvalue = 0;
1405         (void) cv_reltimedwait(&mpt->m_fw_cv, &mpt->m_mutex,
1406             drv_sectohz(60), TR_CLOCK_TICK);
1406             drv_usectohz(60 * MICROSEC), TR_CLOCK_TICK);
1407         if (!(cmd->cmd_flags & CFLAG_FINISHED)) {
1408                 mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1409                 if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
1410                         mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1411                 }
1412                 rvalue = -1;
1413         }
```

```
1414         mptsas_remove_cmd(mpt, cmd);
1415         mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);

1417         return (rvalue);
1418 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  246129 Wed Aug 19 07:25:13 2015
new/usr/src/uts/common/io/scsi/adapters/scsi_vhci/scsi_vhci.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

/* do a PGR out with the information we've saved away */
static int
vhci_do_prout(scsi_vhci_priv_t *svp)
{

        struct scsi_pkt                 *new_pkt;
        struct buf                      *bp;
        scsi_vhci_lun_t                 *vlun = svp->svp_svl;
        int                             rval, retry, nr_retry, ua_retry;
        uint8_t                         *sns, skey;

        bp = getrbuf(KM_SLEEP);
        bp->b_flags = B_WRITE;
        bp->b_resid = 0;
        bp->b_un.b_addr = (caddr_t)&vlun->svl_prout;
        bp->b_bcount = vlun->svl_bcount;

        VHCI_INCR_PATH_CMDCOUNT(svp);

        new_pkt = scsi_init_pkt(&svp->svp_psd->sd_address, NULL, bp,
            CDB_GROUP1, sizeof (struct scsi_arq_status), 0, 0,
            SLEEP_FUNC, NULL);
        if (new_pkt == NULL) {
                VHCI_DECR_PATH_CMDCOUNT(svp);
                freerbuf(bp);
                cmn_err(CE_WARN, "!vhci_do_prout: scsi_init_pkt failed");
                return (0);
        }
        mutex_enter(&vlun->svl_mutex);
        bp->b_un.b_addr = (caddr_t)&vlun->svl_prout;
        bp->b_bcount = vlun->svl_bcount;
        bcopy(vlun->svl_cdb, new_pkt->pkt_cdbp,
            sizeof (vlun->svl_cdb));
        new_pkt->pkt_time = vlun->svl_time;
        mutex_exit(&vlun->svl_mutex);
        new_pkt->pkt_flags = FLAG_NOINTR;

        ua_retry = nr_retry = retry = 0;
again:
        rval = vhci_do_scsi_cmd(new_pkt);
        if (rval != 1) {
                if ((new_pkt->pkt_reason == CMD_CMPLT) &&
                    (SCBP_C(new_pkt) == STATUS_CHECK) &&
                    (new_pkt->pkt_state & STATE_ARQ_DONE)) {
                        sns = (uint8_t *)
                            &(((struct scsi_arq_status *)(uintptr_t)
                            (new_pkt->pkt_scbp))->sts_sensedata);
                        skey = scsi_sense_key(sns);
                        if ((skey == KEY_UNIT_ATTENTION) ||
                            (skey == KEY_NOT_READY)) {
                                int max_retry;
                                struct scsi_failover_ops *fops;
                                fops = vlun->svl_fops;
                                rval = fops->sfo_analyze_sense(svp->svp_psd,
                                    sns, vlun->svl_fops_ctpriv);
                                if (rval == SCSI_SENSE_NOT_READY) {
                                        max_retry = vhci_prout_not_ready_retry;
                                        retry = nr_retry++;
                                        delay(drv_sectohz(1));
```



```
                                        delay(1*drv_usectohz(1000000));
                                } else {
                                        /* chk for state change and update */
                                        if (rval == SCSI_SENSE_STATE_CHANGED) {
                                                int held;
                                                VHCI_HOLD_LUN(vlun,
                                                    VH_NOSLEEP, held);
                                                if (!held) {
                                                        rval = TRAN_BUSY;
                                                } else {
                                                        /* chk for alua first */
                                                        vhci_update_pathstates(
                                                            (void *)vlun);
                                                }
                                        }
                                        retry = ua_retry++;
                                        max_retry = VHCI_MAX_PGR_RETRIES;
                                }
                                if (retry < max_retry) {
                                        VHCI_DEBUG(4, (CE_WARN, NULL,
                                            "!vhci_do_prout retry 0x%x "
                                            "(0x%x 0x%x 0x%x)",
                                            SCBP_C(new_pkt),
                                            new_pkt->pkt_cdbp[0],
                                            new_pkt->pkt_cdbp[1],
                                            new_pkt->pkt_cdbp[2]));
                                        goto again;
                                }
                                rval = 0;
                                VHCI_DEBUG(4, (CE_WARN, NULL,
                                    "!vhci_do_prout 0x%x "
                                    "(0x%x 0x%x 0x%x)",
                                    SCBP_C(new_pkt),
                                    new_pkt->pkt_cdbp[0],
                                    new_pkt->pkt_cdbp[1],
                                    new_pkt->pkt_cdbp[2]));
                        } else if (skey == KEY_ILLEGAL_REQUEST)
                                rval = VHCI_PGR_ILLEGALOP;
                }
        } else {
                rval = 1;
        }
        scsi_destroy_pkt(new_pkt);
        VHCI_DECR_PATH_CMDCOUNT(svp);
        freerbuf(bp);
        return (rval);
}
______unchanged_portion_omitted_

/*
 * path offline handler.  Release all bindings that will not be
 * released by the normal packet transport/completion code path.
 * Since we don't (presently) keep any bindings alive outside of
 * the in-transport packets (which will be released on completion)
 * there is not much to do here.
 */
/* ARGSUSED */
static int
vhci_pathinfo_offline(dev_info_t *vdip, mdi_pathinfo_t *pip, int flags)
{
        scsi_hba_tran_t         *hba = NULL;
        struct scsi_device      *psd = NULL;
        dev_info_t              *pdip = NULL;
        dev_info_t              *cdip = NULL;
        scsi_vhci_priv_t        *svp = NULL;
```

```
5298         ASSERT(vdip != NULL);
5299         ASSERT(pip != NULL);

5301         pdip = mdi_pi_get_phci(pip);
5302         ASSERT(pdip != NULL);
5303         if (pdip == NULL) {
5304                 VHCI_DEBUG(1, (CE_WARN, vdip, "Invalid path 0x%p: NULL "
5305                     "phci dip", (void *)pip));
5306                 return (MDI_FAILURE);
5307         }

5309         cdip = mdi_pi_get_client(pip);
5310         ASSERT(cdip != NULL);
5311         if (cdip == NULL) {
5312                 VHCI_DEBUG(1, (CE_WARN, vdip, "Invalid path 0x%p: NULL "
5313                     "client dip", (void *)pip));
5314                 return (MDI_FAILURE);
5315         }

5317         hba = ddi_get_driver_private(pdip);
5318         ASSERT(hba != NULL);

5320         svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
5321         if (svp == NULL) {
5322                 /*
5323                  * mdi_pathinfo node in INIT state can have vHCI private
5324                  * information set to null
5325                  */
5326                 VHCI_DEBUG(1, (CE_NOTE, vdip, "!vhci_pathinfo_offline: "
5327                     "svp is NULL for pip 0x%p\n", (void *)pip));
5328                 return (MDI_SUCCESS);
5329         }

5331         psd = svp->svp_psd;
5332         ASSERT(psd != NULL);

5334         mutex_enter(&svp->svp_mutex);

5336         VHCI_DEBUG(1, (CE_NOTE, vdip, "!vhci_pathinfo_offline: "
5337             "%d cmds pending on path: 0x%p\n", svp->svp_cmds, (void *)pip));
5338         while (svp->svp_cmds != 0) {
5339                 if (cv_reltimedwait(&svp->svp_cv, &svp->svp_mutex,
5340                     drv_sectohz(vhci_path_quiesce_timeout),
5340                     drv_usectohz(vhci_path_quiesce_timeout * 1000000),
5341                     TR_CLOCK_TICK) == -1) {
5342                         /*
5343                          * The timeout time reached without the condition
5344                          * being signaled.
5345                          */
5346                         VHCI_DEBUG(1, (CE_NOTE, vdip, "!vhci_pathinfo_offline: "
5347                             "Timeout reached on path 0x%p without the cond\n",
5348                             (void *)pip));
5349                         VHCI_DEBUG(1, (CE_NOTE, vdip, "!vhci_pathinfo_offline: "
5350                             "%d cmds still pending on path: 0x%p\n",
5351                             svp->svp_cmds, (void *)pip));
5352                         break;
5353                 }
5354         }
5355         mutex_exit(&svp->svp_mutex);

5357         /*
5358          * Check to see if this vlun has an active SCSI-II RESERVE. And this
5359          * is the pip for the path that has been reserved.
5360          * If so clear the reservation by sending a reset, so the host will not
5361          * get a reservation conflict.  Reset the flag VLUN_RESERVE_ACTIVE_FLG
5362          * for this lun.  Also a reset notify is sent to the target driver
```



```
5363          * just in case the POR check condition is cleared by some other layer
5364          * in the stack.
5365          */
5366         if (svp->svp_svl->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
5367                 if (pip == svp->svp_svl->svl_resrv_pip) {
5368                         if (vhci_recovery_reset(svp->svp_svl,
5369                             &svp->svp_psd->sd_address, TRUE,
5370                             VHCI_DEPTH_TARGET) == 0) {
5371                                 VHCI_DEBUG(1, (CE_NOTE, NULL,
5372                                     "!vhci_pathinfo_offline (pip:%p):"
5373                                     "reset failed, retrying\n", (void *)pip));
5374                                 delay(drv_sectohz(1));
5374                                 delay(1*drv_usectohz(1000000));
5375                                 if (vhci_recovery_reset(svp->svp_svl,
5376                                     &svp->svp_psd->sd_address, TRUE,
5377                                     VHCI_DEPTH_TARGET) == 0) {
5378                                         VHCI_DEBUG(1, (CE_NOTE, NULL,
5379                                             "!vhci_pathinfo_offline "
5380                                             "(pip:%p): reset failed, "
5381                                             "giving up!\n", (void *)pip));
5382                                 }
5383                         }
5384                         svp->svp_svl->svl_flags &= ~VLUN_RESERVE_ACTIVE_FLG;
5385                 }
5386         }

5388         mdi_pi_set_state(pip, MDI_PATHINFO_STATE_OFFLINE);
5389         vhci_mpapi_set_path_state(vdip, pip, MP_DRVR_PATH_STATE_REMOVED);

5391         VHCI_DEBUG(1, (CE_NOTE, NULL,
5392             "!vhci_pathinfo_offline: offlined path 0x%p\n", (void *)pip));
5393         return (MDI_SUCCESS);
5394 }
```

*_____unchanged_portion_omitted_*

```
7487 static int
7488 vhci_quiesce_lun(struct scsi_vhci_lun *vlun)
7489 {
7490         mdi_pathinfo_t          *pip, *spip;
7491         dev_info_t              *cdip;
7492         struct scsi_vhci_priv   *svp;
7493         mdi_pathinfo_state_t    pstate;
7494         uint32_t                p_ext_state;
7495         int                     circular;

7497         cdip = vlun->svl_dip;
7498         pip = spip = NULL;
7499         ndi_devi_enter(cdip, &circular);
7500         pip = mdi_get_next_phci_path(cdip, NULL);
7501         while (pip != NULL) {
7502                 (void) mdi_pi_get_state2(pip, &pstate, &p_ext_state);
7503                 if (pstate != MDI_PATHINFO_STATE_ONLINE) {
7504                         spip = pip;
7505                         pip = mdi_get_next_phci_path(cdip, spip);
7506                         continue;
7507                 }
7508                 mdi_hold_path(pip);
7509                 ndi_devi_exit(cdip, circular);
7510                 svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
7511                 mutex_enter(&svp->svp_mutex);
7512                 while (svp->svp_cmds != 0) {
7513                         if (cv_reltimedwait(&svp->svp_cv, &svp->svp_mutex,
7514                             drv_sectohz(vhci_path_quiesce_timeout),
7514                             drv_usectohz(vhci_path_quiesce_timeout * 1000000),
7515                             TR_CLOCK_TICK) == -1) {
7516                                 mutex_exit(&svp->svp_mutex);
```

```
7517                                mdi_rele_path(pip);
7518                                VHCI_DEBUG(1, (CE_WARN, NULL,
7519                                    "Quiesce of lun is not successful "
7520                                    "vlun: 0x%p.", (void *)vlun));
7521                                return (0);
7522                        }
7523                }
7524                mutex_exit(&svp->svp_mutex);
7525                ndi_devi_enter(cdip, &circular);
7526                spip = pip;
7527                pip = mdi_get_next_phci_path(cdip, spip);
7528                mdi_rele_path(spip);
7529        }
7530        ndi_devi_exit(cdip, circular);
7531        return (1);
7532 }
```

*unchanged_portion_omitted_*

```
8306 /*
8307  * auto-rqsense is not enabled so we have to retrieve the request sense
8308  * manually.
8309  */
8310 static int
8311 vhci_uscsi_send_sense(struct scsi_pkt *pkt, mp_uscsi_cmd_t *mp_uscmdp)
8312 {
8313        struct buf              *rqbp, *cmdbp;
8314        struct scsi_pkt         *rqpkt;
8315        int                     rval = 0;

8317        cmdbp = mp_uscmdp->cmdbp;
8318        ASSERT(cmdbp != NULL);

8320        VHCI_DEBUG(4, (CE_WARN, NULL,
8321            "vhci_uscsi_send_sense: enter: bp: %p pkt: %p scmd: %p",
8322            (void *)cmdbp, (void *)pkt, (void *)mp_uscmdp));
8323        /* set up the packet information and cdb */
8324        if ((rqbp = scsi_alloc_consistent_buf(mp_uscmdp->ap, NULL,
8325            SENSE_LENGTH, B_READ, NULL, NULL)) == NULL) {
8326                return (-1);
8327        }

8329        if ((rqpkt = scsi_init_pkt(mp_uscmdp->ap, NULL, rqbp,
8330            CDB_GROUP0, 1, 0, PKT_CONSISTENT, NULL, NULL)) == NULL) {
8331                scsi_free_consistent_buf(rqbp);
8332                return (-1);
8333        }

8335        (void) scsi_setup_cdb((union scsi_cdb *)(intptr_t)rqpkt->pkt_cdbp,
8336            SCMD_REQUEST_SENSE, 0, SENSE_LENGTH, 0);

8338        mp_uscmdp->rqbp = rqbp;
8339        rqbp->b_private = mp_uscmdp;
8340        rqpkt->pkt_flags |= FLAG_SENSING;
8341        rqpkt->pkt_time = 60;
8342        rqpkt->pkt_comp = vhci_uscsi_iodone;
8343        rqpkt->pkt_private = mp_uscmdp;

8345        /*
8346         * NOTE: This code path is related to MPAPI uscsi(7I), so path
8347         * selection is not based on path_instance.
8348         */
8349        if (scsi_pkt_allocated_correctly(rqpkt))
8350                rqpkt->pkt_path_instance = 0;

8352        /* get her done */
8353        switch (scsi_transport(rqpkt)) {
```



```
8354        case TRAN_ACCEPT:
8355                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_uscsi_send_sense: "
8356                    "transport accepted."));
8357                break;
8358        case TRAN_BUSY:
8359                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_uscsi_send_sense: "
8360                    "transport busy, setting timeout."));
8361                vhci_restart_timeid = timeout(vhci_uscsi_restart_sense, rqpkt,
8362                    drv_sectohz(5));
8362                    (drv_usectohz(5 * 1000000)));
8363                break;
8364        default:
8365                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_uscsi_send_sense: "
8366                    "transport failed"));
8367                scsi_free_consistent_buf(rqbp);
8368                scsi_destroy_pkt(rqpkt);
8369                rval = -1;
8370        }

8372        return (rval);
8373 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  316105 Wed Aug 19 07:25:14 2015
new/usr/src/uts/common/io/scsi/impl/scsi_hba.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

5680 /*
5681  * Add an entry to the list of barrier nodes to be asynchronously deleted by
5682  * the scsi_hba_barrier_daemon after the specified timeout. Nodes on
5683  * the barrier list are used to implement the bus_config probe cache
5684  * of non-existent devices. The nodes are at DS_INITIALIZED, so their
5685  * @addr is established for searching. Since devi_ref of a DS_INITIALIZED
5686  * node will *not* prevent demotion, demotion is prevented by setting
5687  * sd_uninit_prevent. Devinfo snapshots attempt to attach probe cache
5688  * nodes, and on failure attempt to demote the node (without the participation
5689  * of bus_unconfig) to DS_BOUND - this demotion is prevented via
5690  * sd_uninit_prevent causing any attempted DDI_CTLOPS_UNINITCHILD to fail.
5691  * Probe nodes are bound to nulldriver. The list is sorted by
5692  * expiration time.
5693  *
5694  * NOTE: If we drove a probe node to DS_ATTACHED, we could use ndi_hold_devi()
5695  * to prevent demotion (instead of sd_uninit_prevent).
5696  */
5697 static void
5698 scsi_hba_barrier_add(dev_info_t *probe, int seconds)
5699 {
5700         struct scsi_hba_barrier *nb;
5701         struct scsi_hba_barrier *b;
5702         struct scsi_hba_barrier **bp;
5703         clock_t                 endtime;

5705         ASSERT(scsi_hba_devi_is_barrier(probe));

5707         /* HBA is no longer responsible for nodes on the barrier list. */
5708         scsi_hba_barrier_tran_tgt_free(probe);
5709         nb = kmem_alloc(sizeof (struct scsi_hba_barrier), KM_SLEEP);
5710         mutex_enter(&scsi_hba_barrier_mutex);
5711         endtime = ddi_get_lbolt() + drv_sectohz(seconds);
5711         endtime = ddi_get_lbolt() + drv_usectohz(seconds * MICROSEC);
5712         for (bp = &scsi_hba_barrier_list; (b = *bp) != NULL;
5713             bp = &b->barrier_next)
5714                 if (b->barrier_endtime > endtime)
5715                         break;
5716         nb->barrier_next = *bp;
5717         nb->barrier_endtime = endtime;
5718         nb->barrier_probe = probe;
5719         *bp = nb;
5720         if (bp == &scsi_hba_barrier_list)
5721                 (void) cv_signal(&scsi_hba_barrier_cv);
5722         mutex_exit(&scsi_hba_barrier_mutex);
5723 }
______unchanged_portion_omitted_

6008 /*
6009  * Enumerate a child at the specified @addr. If a device exists @addr then
6010  * ensure that we have the appropriately named devinfo node for it. Name is
6011  * NULL in the bus_config_all case. This routine has no knowledge of the
6012  * format of an @addr string or associated addressing properties.
6013  *
6014  * The caller must guarantee that there is an open scsi_hba_devi_enter on the
6015  * parent. We return the scsi_device structure for the child device. This
6016  * scsi_device structure is valid until the caller scsi_hba_devi_exit the
6017  * parent. The caller can add do ndi_hold_devi of the child prior to the
6018  * scsi_hba_devi_exit to extend the validity of the child.
6019  *
```



```
6020  * In some cases the returned scsi_device structure may be used to drive
6021  * additional SCMD_REPORT_LUNS operations by bus_config_all callers.
6022  *
6023  * The first operation performed is to see if there is a dynamic SID nodes
6024  * already attached at the specified "name@addr". This is the fastpath
6025  * case for resolving a reference to a node that has already been created.
6026  * All other references are serialized for a given @addr prior to probing
6027  * to determine the type of device, if any, at the specified @addr.
6028  * If no device is present then NDI_FAILURE is returned. The fact that a
6029  * device does not exist may be determined via the barrier/probe cache,
6030  * minimizing the probes of non-existent devices.
6031  *
6032  * When there is a device present the dynamic SID node is created based on
6033  * the device found. If a driver.conf node exists for the same @addr it
6034  * will either merge into the dynamic SID node (if the SID node bound to
6035  * that driver), or exist independently. To prevent the actions of one driver
6036  * causing side effects in another, code prevents multiple SID nodes from
6037  * binding to the same "@addr" at the same time. There is autodetach code
6038  * to allow one device to be replaced with another at the same @addr for
6039  * slot addressed SCSI bus implementations (SPI). For compatibility with
6040  * legacy driver.conf behavior, the code does not prevent multiple driver.conf
6041  * nodes from attaching to the same @addr at the same time.
6042  *
6043  * This routine may have the side effect of creating nodes for devices other
6044  * than the one being sought. It is possible that there is a different type of
6045  * target device at that target/lun address than we were asking for. In that
6046  * It is the caller's responsibility to determine whether the device we found,
6047  * if any, at the specified address, is the one it really wanted.
6048  */
6049 static struct scsi_device *
6050 scsi_device_config(dev_info_t *self, char *name, char *addr, scsi_enum_t se,
6051     int *circp, int *ppi)
6052 {
6053         dev_info_t              *child = NULL;
6054         dev_info_t              *probe = NULL;
6055         struct scsi_device      *sdchild;
6056         struct scsi_device      *sdprobe;
6057         dev_info_t              *dsearch;
6058         mdi_pathinfo_t          *psearch;
6059         major_t                 major;
6060         int                     sp;
6061         int                     pi = 0;
6062         int                     wait_msg = scsi_hba_wait_msg;
6063         int                     chg;

6065         ASSERT(self && addr && DEVI_BUSY_OWNED(self));

6067         SCSI_HBA_LOG((_LOG(4), self, NULL, "%s@%s wanted",
6068             name ? name : "", addr));

6070         /* playing with "probe" node name is dangerous */
6071         if (name && (strcmp(name, "probe") == 0))
6072                 return (NULL);

6074         /*
6075          * NOTE: use 'goto done;' or 'goto fail;'. There should only be one
6076          * 'return' statement from here to the end of the function - the one
6077          * on the last line of the function.
6078          */

6080         /*
6081          * Fastpath: search to see if we are requesting a named SID node that
6082          * already exists (we already created) - probe node does not count.
6083          * scsi_findchild() does not hold the returned devinfo node, but
6084          * this is OK since the caller has a scsi_hba_devi_enter on the
6085          * attached parent HBA (self). The caller is responsible for attaching
```

```
6086          * and placing a hold on the child (directly via ndi_hold_devi or
6087          * indirectly via ndi_busop_bus_config) before doing an
6088          * scsi_hba_devi_exit on the parent.
6089          *
6090          * NOTE: This fastpath prevents detecting a driver binding change
6091          * (autodetach) if the same nodename is used for old and new binding.
6092          */
6093         /* first call is with init set */
6094         (void) scsi_findchild(self, name, addr, 1, &dsearch, NULL, &pi);
6095         if (dsearch && scsi_hba_dev_is_sid(dsearch) &&
6096             !scsi_hba_devi_is_barrier(dsearch)) {
6097                 SCSI_HBA_LOG((_LOG(4), NULL, dsearch,
6098                     "%s@%s devinfo fastpath", name ? name : "", addr));
6099                 child = dsearch;
6100                 goto done;
6101         }

6103         /*
6104          * Create a barrier devinfo node used to "probe" the device with. We
6105          * need to drive this node to DS_INITIALIZED so that the
6106          * DDI_CTLOPS_INITCHILD has occurred, bringing the SCSA transport to
6107          * a state useable state for issuing our "probe" commands. We establish
6108          * this barrier node with a node name of "probe" and compatible
6109          * property of "scsiprobe". The compatible property must be associated
6110          * in /etc/driver_aliases with a scsi target driver available in the
6111          * root file system (sd).
6112          *
6113          * The "probe" that we perform on the barrier node, after it is
6114          * DS_INITIALIZED, is used to find the information needed to create a
6115          * dynamic devinfo (SID) node. This "probe" is separate from the
6116          * probe(9E) call associated with the transition of a node from
6117          * DS_INITIALIZED to DS_PROBED. The probe(9E) call that eventually
6118          * occurs against the created SID node should find ddi_dev_is_sid and
6119          * just return DDI_PROBE_DONTCARE.
6120          *
6121          * Trying to avoid the use of a barrier node is not a good idea
6122          * because we may have an HBA driver that uses generic bus_config
6123          * (this code) but implements its own DDI_CTLOPS_INITCHILD with side
6124          * effects that we can't duplicate (such as the ATA nexus driver).
6125          *
6126          * The probe/barrier node plays an integral part of the locking scheme.
6127          * The objective is to single thread probes of the same device (same
6128          * @addr) while allowing parallelism for probes of different devices
6129          * with the same parent. At this point we are serialized on our self.
6130          * For parallelism we will need to release our self. Prior to release
6131          * we construct a barrier for probes of the same device to serialize
6132          * against. The "probe@addr" node acts as this barrier. An entering
6133          * thread must wait until the probe node does not exist - it can then
6134          * create and link the probe node - dropping the HBA (self) lock after
6135          * the node is linked and visible (after ddi_initchild). A side effect
6136          * of this is that transports should not "go over the wire" (i.e. do
6137          * things that incur significant delays) until after tran_target_init.
6138          * This means that the first "over the wire" operation should occur
6139          * at tran_target_probe time - when things are running in parallel
6140          * again.
6141          *
6142          * If the probe node exists then another probe with the same @addr is
6143          * in progress, we must wait until there is no probe in progress
6144          * before proceeding, and when we proceed we must continue to hold the
6145          * HBA (self) until we have linked a new probe node as a barrier.
6146          *
6147          * When a device is found to *not* exist, its probe/barrier node may be
6148          * marked with DEVICE_REMOVED with node deletion scheduled for some
6149          * future time (seconds). This asynchronous deletion allows the
6150          * framework to detect repeated requests to the same non-existent
6151          * device and avoid overhead associated with contacting a non-existent
```



```
6152          * device again and again.
6153          */
6154         for (;;) {
6155                 /*
6156                  * Search for probe node - they should only exist as devinfo
6157                  * nodes.
6158                  */
6159                 (void) scsi_findchild(self, "probe", addr,
6160                     0, &probe, &psearch, NULL);
6161                 if (probe == NULL) {
6162                         if (psearch)
6163                                 SCSI_HBA_LOG((_LOG(2), self,
6164                                     mdi_pi_get_client(psearch),
6165                                     "???? @%s 'probe' search found "
6166                                     "pathinfo: %p", addr, (void *)psearch));
6167                         break;
6168                 }

6170                 /*
6171                  * The barrier node may cache the non-existence of a device
6172                  * by leaving the barrier node in place (with
6173                  * DEVI_DEVICE_REMOVED flag set ) for some amount of time after
6174                  * the failure of a probe. This flag is used to fail
6175                  * additional probes until the barrier probe node is deleted,
6176                  * which will occur from a timeout some time after a failed
6177                  * probe. The failed probe will use DEVI_SET_DEVICE_REMOVED
6178                  * and schedule probe node deletion from a timeout. The callers
6179                  * scsi_hba_devi_exit on the way out of the first failure will
6180                  * do the cv_broadcast associated with the cv_wait below - this
6181                  * handles threads that wait prior to DEVI_DEVICE_REMOVED being
6182                  * set.
6183                  */
6184                 if (DEVI_IS_DEVICE_REMOVED(probe)) {
6185                         SCSI_HBA_LOG((_LOG(3), NULL, probe,
6186                             "detected probe DEVICE_REMOVED"));
6187                         probe = NULL;   /* deletion already scheduled */
6188                         goto fail;
6189                 }

6191                 /*
6192                  * Drop the lock on the HBA (self) and wait until the probe in
6193                  * progress has completed. A changes in the sibling list from
6194                  * removing the probe node will cause cv_wait to return
6195                  * (scsi_hba_devi_exit does the cv_broadcast).
6196                  */
6197                 if (wait_msg) {
6198                         wait_msg--;
6199                         SCSI_HBA_LOG((_LOG(2), NULL, probe,
6200                             "exists, probe already in progress: %s", wait_msg ?
6201                             "waiting..." : "last msg, but still waiting..."));
6202                 }

6204                 /*
6205                  * NOTE: we could avoid rare case of one second delay by
6206                  * implementing scsi_hba_devi_exit_and_wait based on
6207                  * ndi/mdi_devi_exit_and_wait (and consider switching devcfg.c
6208                  * code to use these ndi/mdi interfaces too).
6209                  */
6210                 scsi_hba_devi_exit(self, *circp);
6211                 mutex_enter(&DEVI(self)->devi_lock);
6212                 (void) cv_timedwait(&DEVI(self)->devi_cv,
6213                     &DEVI(self)->devi_lock,
6214                     ddi_get_lbolt() + drv_sectohz(1));
6214                     ddi_get_lbolt() + drv_usectohz(MICROSEC));
6215                 mutex_exit(&DEVI(self)->devi_lock);
6216                 scsi_hba_devi_enter(self, circp);
```

```
6217         }
6218         ASSERT(probe == NULL);

6220         /*
6221          * Search to see if we are requesting a SID node that already exists.
6222          * We hold the HBA (self) and there is not another probe in progress at
6223          * the same @addr. scsi_findchild() does not hold the returned
6224          * devinfo node but this is OK since we hold the HBA (self).
6225          */
6226         if (name) {
6227                 (void) scsi_findchild(self, name, addr, 1, &dsearch, NULL, &pi);
6228                 if (dsearch && scsi_hba_dev_is_sid(dsearch)) {
6229                         SCSI_HBA_LOG((_LOG(4), NULL, dsearch,
6230                             "%s@%s probe devinfo fastpath",
6231                             name ? name : "", addr));
6232                         child = dsearch;
6233                         goto done;
6234                 }
6235         }

6237         /*
6238          * We are looking for a SID node that does not exist or a driver.conf
6239          * node.
6240          *
6241          * To avoid probe side effects, before we probe the device at the
6242          * specified address we need to check to see if there is already an
6243          * initialized child "@addr".
6244          *
6245          * o If we find an initialized SID child and name is NULL or matches
6246          *   the name or the name of the attached driver then we return the
6247          *   existing node.
6248          *
6249          * o If we find a non-matching SID node, we will attempt to autodetach
6250          *   and remove the node in preference to our new node.
6251          *
6252          * o If SID node found does not match and can't be autodetached, we
6253          *   fail: we only allow one SID node at an address.
6254          *
6255          * NOTE: This code depends on SID nodes showing up prior to
6256          * driver.conf nodes in the sibling list.
6257          */
6258         for (;;) {
6259                 /* first NULL name call is with init set */
6260                 (void) scsi_findchild(self, NULL, addr, 1, &dsearch, NULL, &pi);
6261                 if (dsearch == NULL)
6262                         break;
6263                 ASSERT(!scsi_hba_devi_is_barrier(dsearch));

6265                 /*
6266                  * To detect changes in driver binding that should attempt
6267                  * autodetach we determine the major number of the driver
6268                  * that should currently be associated with the device based
6269                  * on the compatible property.
6270                  */
6271                 major = DDI_MAJOR_T_NONE;
6272                 if (scsi_hba_dev_is_sid(dsearch))
6273                         major = ddi_compatible_driver_major(dsearch, NULL);
6274                 if ((major == DDI_MAJOR_T_NONE) && (name == NULL))
6275                         major = ddi_driver_major(dsearch);

6277                 if ((scsi_hba_dev_is_sid(dsearch) ||
6278                     (i_ddi_node_state(dsearch) >= DS_INITIALIZED)) &&
6279                     ((name == NULL) ||
6280                     (strcmp(ddi_node_name(dsearch), name) == 0) ||
6281                     (strcmp(ddi_driver_name(dsearch), name) == 0)) &&
6282                     (major == ddi_driver_major(dsearch))) {
```



```
6283                         SCSI_HBA_LOG((_LOG(3), NULL, dsearch,
6284                             "already attached @addr"));
6285                         child = dsearch;
6286                         goto done;
6287                 }

6289                 if (!scsi_hba_dev_is_sid(dsearch))
6290                         break;                  /* driver.conf node */

6292                 /*
6293                  * Implement autodetach of SID node for situations like a
6294                  * previously "scsinodev" LUN0 coming into existence (or a
6295                  * disk/tape on an SPI transport at same addr but never both
6296                  * powered on at the same time). Try to autodetach the existing
6297                  * SID node @addr. If that works, search again - otherwise fail.
6298                  */
6299                 SCSI_HBA_LOG((_LOG(2), NULL, dsearch,
6300                     "looking for %s@%s: SID @addr exists, autodetach",
6301                     name ? name : "", addr));
6302                 if (!scsi_hba_remove_node(dsearch)) {
6303                         SCSI_HBA_LOG((_LOG(2), NULL, dsearch,
6304                             "autodetach @%s failed: fail %s@%s",
6305                             addr, name ? name : "", addr));
6306                         goto fail;
6307                 }
6308                 SCSI_HBA_LOG((_LOG(2), self, NULL, "autodetach @%s OK", addr));
6309         }

6311         /*
6312          * We will be creating a new SID node, allocate probe node
6313          * used to find out information about the device located @addr.
6314          * The probe node also acts as a barrier against additional
6315          * configuration at the same address, and in the case of non-existent
6316          * devices it will (for some amount of time) avoid re-learning that
6317          * the device does not exist on every reference. Once the probe
6318          * node is DS_LINKED we can drop the HBA (self).
6319          *
6320          * The probe node is allocated as a hidden node so that it does not
6321          * show up in devinfo snapshots.
6322          */
6323         ndi_devi_alloc_sleep(self, "probe",
6324             (se == SE_HP) ? DEVI_SID_HP_HIDDEN_NODEID : DEVI_SID_HIDDEN_NODEID,
6325             &probe);
6326         ASSERT(probe);
6327         ndi_flavor_set(probe, SCSA_FLAVOR_SCSI_DEVICE);

6329         /*
6330          * Decorate the probe node with the property representation of @addr
6331          * unit-address string prior to initchild so that initchild can
6332          * construct the name of the node from properties and tran_tgt_init
6333          * implementation can determine what LUN is being referenced.
6334          *
6335          * If the addr specified has incorrect syntax (busconfig one of bogus
6336          * /devices path) then scsi_hba_ua_set can fail.  If the address
6337          * is not understood by the SCSA HBA driver then this operation will
6338          * work, but tran_tgt_init may still fail (for example the HBA
6339          * driver may not support secondary functions).
6340          */
6341         if (scsi_hba_ua_set(addr, probe, NULL) == 0) {
6342                 SCSI_HBA_LOG((_LOG(2), NULL, probe,
6343                     "@%s failed scsi_hba_ua_set", addr));
6344                 goto fail;
6345         }

6347         /*
6348          * Set the class property to "scsi". This is sufficient to distinguish
```

```
6349          * the node for HBAs that have multiple classes of children (like uata
6350          * - which has "dada" class for ATA children and "scsi" class for
6351          * ATAPI children) and may not use our scsi_busctl_initchild()
6352          * implementation. We also add a "compatible" property of "scsiprobe"
6353          * to select the probe driver.
6354          */
6355         if ((ndi_prop_update_string(DDI_DEV_T_NONE, probe,
6356             "class", "scsi") != DDI_PROP_SUCCESS) ||
6357             (ndi_prop_update_string_array(DDI_DEV_T_NONE, probe,
6358             "compatible", &compatible_probe, 1) != DDI_PROP_SUCCESS)) {
6359                 SCSI_HBA_LOG((_LOG(1), NULL, probe,
6360                     "@%s failed node decoration", addr));
6361                 goto fail;
6362         }
6363
6364         /*
6365          * Promote probe node to DS_INITIALIZED so that transport can be used
6366          * for scsi_probe. After this the node is linked and visible as a
6367          * barrier for serialization of other @addr operations.
6368          *
6369          * NOTE: If we attached the probe node, we could get rid of
6370          * uninit_prevent.
6371          */
6372         if (ddi_initchild(self, probe) != DDI_SUCCESS) {
6373                 SCSI_HBA_LOG((_LOG(2), NULL, probe,
6374                     "@%s failed initchild", addr));
6375
6376                 /* probe node will be removed in fail exit path */
6377                 goto fail;
6378         }
6379
6380         /* get the scsi_device structure of the probe node */
6381         sdprobe = ddi_get_driver_private(probe);
6382         ASSERT(sdprobe);
6383
6384         /*
6385          * Do scsi_probe. The probe node is linked and visible as a barrier.
6386          * We prevent uninitialization of the probe node and drop our HBA (self)
6387          * while we run scsi_probe() of this "@addr". This allows the framework
6388          * to support multiple scsi_probes for different devices attached to
6389          * the same HBA (self) in parallel. We prevent node demotion of the
6390          * probe node from DS_INITIALIZED by setting sd_uninit_prevent. The
6391          * probe node can not be successfully demoted below DS_INITIALIZED
6392          * (scsi_busctl_uninitchild will fail) until we zero sd_uninit_prevent
6393          * as we are freeing the node via scsi_hba_remove_node(probe).
6394          */
6395         sdprobe->sd_uninit_prevent++;
6396         scsi_hba_devi_exit(self, *circp);
6397         sp = scsi_probe(sdprobe, SLEEP_FUNC);
6398
6399         /* Introduce a small delay here to increase parallelism. */
6400         delay_random(5);
6401
6402         if (sp == SCSIPROBE_EXISTS) {
6403                 /*
6404                  * For a device that exists, while still running in parallel,
6405                  * also get identity information from device. This is done
6406                  * separate from scsi_probe/tran_tgt_probe/scsi_hba_probe
6407                  * since the probe code path may still be used for HBAs
6408                  * that don't use common bus_config services (we don't want
6409                  * to expose that code path to a behavior change). This
6410                  * operation is called 'identity' to avoid confusion with
6411                  * deprecated identify(9E).
6412                  *
6413                  * Future: We may eventually want to allow HBA customization via
6414                  * scsi_identity/tran_tgt_identity/scsi_device_identity, but for
```



```
6415                  * now we just scsi_device_identity.
6416                  *
6417                  * The identity operation will establish additional properties
6418                  * on the probe node related to device identity:
6419                  *
6420                  *      "inquiry-page-80"       byte array of SCSI page 80
6421                  *      "inquiry-page-83"       byte array of SCSI page 83
6422                  *
6423                  * These properties will be used to generate a devid
6424                  * (ddi_devid_scsi_encode) and guid - and to register
6425                  * (ddi_devid_register) a devid for the device.
6426                  *
6427                  * If identify fails (non-zero return), the we had allocation
6428                  * problems or the device returned inconsistent results then
6429                  * we pretend that device does not exist.
6430                  */
6431                 if (scsi_device_identity(sdprobe, SLEEP_FUNC)) {
6432                         scsi_enumeration_failed(probe, -1, NULL, "identify");
6433                         sp = SCSIPROBE_FAILURE;
6434                 }
6435
6436                 /*
6437                  * Future: Is there anything more we can do here to help avoid
6438                  * serialization on iport parent during scsi_device attach(9E)?
6439                  */
6440         }
6441         scsi_hba_devi_enter(self, circp);
6442         sdprobe->sd_uninit_prevent--;
6443
6444         if (sp != SCSIPROBE_EXISTS) {
6445                 scsi_enumeration_failed(probe, -1, NULL, "probe");
6446
6447                 if ((se != SE_HP) && scsi_hba_barrier_timeout) {
6448                         /*
6449                          * Target does not exist. Mark the barrier probe node
6450                          * as DEVICE_REMOVED and schedule an asynchronous
6451                          * deletion of the node in scsi_hba_barrier_timeout
6452                          * seconds. We keep our hold on the probe node
6453                          * until we are ready perform the asynchronous node
6454                          * deletion.
6455                          */
6456                         SCSI_HBA_LOG((_LOG(3), NULL, probe,
6457                             "set probe DEVICE_REMOVED"));
6458                         mutex_enter(&DEVI(probe)->devi_lock);
6459                         DEVI_SET_DEVICE_REMOVED(probe);
6460                         mutex_exit(&DEVI(probe)->devi_lock);
6461
6462                         scsi_hba_barrier_add(probe, scsi_hba_barrier_timeout);
6463                         probe = NULL;
6464                 }
6465                 goto fail;
6466         }
6467
6468         /* Create the child node from the inquiry data in the probe node. */
6469         if ((child = scsi_device_configchild(self, addr, se, sdprobe,
6470             circp, &pi)) == NULL) {
6471                 /*
6472                  * This may fail because there was no driver binding identified
6473                  * via driver_alias. We may still have a conf node.
6474                  */
6475                 if (name) {
6476                         (void) scsi_findchild(self, name, addr,
6477                             0, &child, NULL, &pi);
6478                         if (child)
6479                                 SCSI_HBA_LOG((_LOG(2), NULL, child,
6480                                     "using driver.conf driver binding"));
```

```
6481                }
6482                if (child == NULL) {
6483                        SCSI_HBA_LOG((_LOG(2), NULL, probe,
6484                            "device not configured"));
6485                        goto fail;
6486                }
6487        }

6489        /*
6490         * Transfer the inquiry data from the probe node to the child
6491         * SID node to avoid an extra scsi_probe. Callers depend on
6492         * established inquiry data for the returned scsi_device.
6493         */
6494        sdchild = ddi_get_driver_private(child);
6495        if (sdchild && (sdchild->sd_inq == NULL)) {
6496                sdchild->sd_inq = sdprobe->sd_inq;
6497                sdprobe->sd_inq = NULL;
6498        }

6500        /*
6501         * If we are doing a bus_configone and the node we created has the
6502         * wrong node and driver name then switch the return result to a
6503         * driver.conf node with the correct name - if such a node exists.
6504         */
6505        if (name && (strcmp(ddi_node_name(child), name) != 0) &&
6506            (strcmp(ddi_driver_name(child), name) != 0)) {
6507                (void) scsi_findchild(self, name, addr,
6508                    0, &dsearch, NULL, &pi);
6509                if (dsearch == NULL) {
6510                        SCSI_HBA_LOG((_LOG(2), NULL, child,
6511                            "wrong device configured %s@%s", name, addr));
6512                        /*
6513                         * We can't remove when modrootloaded == 0 in case
6514                         * boot-device a uses generic name and
6515                         * scsi_hba_nodename_compatible_get() returned a
6516                         * legacy binding-set driver oriented name.
6517                         */
6518                        if (modrootloaded) {
6519                                (void) scsi_hba_remove_node(child);
6520                                child = NULL;
6521                                goto fail;
6522                        }
6523                } else {
6524                        SCSI_HBA_LOG((_LOG(2), NULL, dsearch,
6525                            "device configured, but switching to driver.conf"));
6526                        child = dsearch;
6527                }
6528        }

6530        /* get the scsi_device structure from the node */
6531        SCSI_HBA_LOG((_LOG(3), NULL, child, "device configured"));

6533        if (child) {
6534 done:          ASSERT(child);
6535                sdchild = ddi_get_driver_private(child);
6536                ASSERT(sdchild);

6538                /*
6539                 * We may have ended up here after promotion of a previously
6540                 * demoted node, where demotion deleted sd_inq data in
6541                 * scsi_busctl_uninitchild.  We redo the scsi_probe() to
6542                 * reestablish sd_inq.  We also want to redo the scsi_probe
6543                 * for devices are currently device_isremove in order to
6544                 * detect new device_insert.
6545                 */
6546                if ((sdchild->sd_inq == NULL) ||
```



```
6547                    ((pi == NULL) && ndi_devi_device_isremoved(child))) {

6549                        /* hotplug_node can only be revived via hotplug. */
6550                        if ((se == SE_HP) || !ndi_dev_is_hotplug_node(child)) {
6551                                SCSI_HBA_LOG((_LOG(3), NULL, child,
6552                                    "scsi_probe() demoted devinfo"));

6554                                sp = scsi_probe(sdchild, SLEEP_FUNC);

6556                                if (sp == SCSIPROBE_EXISTS) {
6557                                        ASSERT(sdchild->sd_inq);

6559                                        /*
6560                                         * Devinfo child exists and we are
6561                                         * talking to the device, report
6562                                         * reinsert and note if this was a
6563                                         * new reinsert.
6564                                         */
6565                                        chg = ndi_devi_device_insert(child);
6566                                        SCSI_HBA_LOG((_LOGCFG, NULL, child,
6567                                            "devinfo %s@%s device_reinsert%s",
6568                                            name ? name : "", addr,
6569                                            chg ? "" : "ed already"));
6570                                } else {
6571                                        scsi_enumeration_failed(child, se,
6572                                            NULL, "reprobe");

6574                                        chg = ndi_devi_device_remove(child);
6575                                        SCSI_HBA_LOG((_LOG(2), NULL, child,
6576                                            "%s device_remove%s",
6577                                            (sp > (sizeof (scsi_probe_ascii) /
6578                                            sizeof (scsi_probe_ascii[0]))) ?
6579                                            "UNKNOWN" : scsi_probe_ascii[sp],
6580                                            chg ? "" : "ed already"));

6582                                        child = NULL;
6583                                        sdchild = NULL;
6584                                }
6585                        } else {
6586                                SCSI_HBA_LOG((_LOG(2), NULL, child,
6587                                    "no reprobe"));

6589                                child = NULL;
6590                                sdchild = NULL;
6591                        }
6592                }
6593        } else {
6594 fail:          ASSERT(child == NULL);
6595                sdchild = NULL;
6596        }
6597        if (probe) {
6598                /*
6599                 * Clean up probe node, destroying node if uninit_prevent
6600                 * it is going to zero. Destroying the probe node (deleting
6601                 * from the sibling list) will wake up any people waiting on
6602                 * the probe node barrier.
6603                 */
6604                SCSI_HBA_LOG((_LOG(4), NULL, probe, "remove probe"));
6605                if (!scsi_hba_remove_node(probe)) {
6606                        /*
6607                         * Probe node removal should not fail, but if it
6608                         * does we hand that responsibility over to the
6609                         * async barrier deletion thread - other references
6610                         * to the same unit-address can hang until the
6611                         * probe node delete completes.
6612                         */
```

```
6613                        SCSI_HBA_LOG((_LOG(4), NULL, probe,
6614                            "remove probe failed, go async"));
6615                        scsi_hba_barrier_add(probe, 1);
6616                }
6617                probe = NULL;
6618        }

6620        /*
6621         * If we successfully resolved via a pathinfo node, we need to find
6622         * the pathinfo node and ensure that it is online (if possible). This
6623         * is done for the case where the device was open when
6624         * scsi_device_unconfig occurred, so mdi_pi_free did not occur. If the
6625         * device has now been reinserted, we want the path back online.
6626         * NOTE: This needs to occur after destruction of the probe node to
6627         * avoid ASSERT related to two nodes at the same unit-address.
6628         */
6629        if (sdchild && pi && (probe == NULL)) {
6630                ASSERT(MDI_PHCI(self));

6632                (void) scsi_findchild(self, NULL, addr,
6633                    0, &dsearch, &psearch, NULL);
6634                ASSERT((psearch == NULL) ||
6635                    (mdi_pi_get_client(psearch) == child));

6637                if (psearch && mdi_pi_device_isremoved(psearch)) {
6638                        /*
6639                         * Verify that we can talk to the device, and if
6640                         * so note if this is a new device_insert.
6641                         *
6642                         * NOTE: We depend on mdi_path_select(), when given
6643                         * a specific path_instance, to select that path
6644                         * even if the path is offline.
6645                         *
6646                         * NOTE: A Client node is not ndi_dev_is_hotplug_node().
6647                         */
6648                        if (se == SE_HP) {
6649                                SCSI_HBA_LOG((_LOG(3), NULL, child,
6650                                    "%s scsi_probe() demoted pathinfo",
6651                                    mdi_pi_spathname(psearch)));

6653                                sp = scsi_hba_probe_pi(sdchild, SLEEP_FUNC, pi);

6655                                if (sp == SCSIPROBE_EXISTS) {
6656                                        /*
6657                                         * Pathinfo child exists and we are
6658                                         * talking to the device, report
6659                                         * reinsert and note if this
6660                                         * was a new reinsert.
6661                                         */
6662                                        chg = mdi_pi_device_insert(psearch);
6663                                        SCSI_HBA_LOG((_LOGCFG, self, NULL,
6664                                            "pathinfo %s device_reinsert%s",
6665                                            mdi_pi_spathname(psearch),
6666                                            chg ? "" : "ed already"));

6668                                        if (chg)
6669                                                (void) mdi_pi_online(psearch,
6670                                                    0);

6672                                        /*
6673                                         * Report client reinsert and note if
6674                                         * this was a new reinsert.
6675                                         */
6676                                        chg = ndi_devi_device_insert(child);
6677                                        SCSI_HBA_LOG((_LOGCFG, NULL, child,
6678                                            "client devinfo %s@%s "
```



```
6679                                            "device_reinsert%s",
6680                                            name ? name : "", addr,
6681                                            chg ? "" : "ed already"));
6682                                } else {
6683                                        scsi_enumeration_failed(child, se,
6684                                            mdi_pi_spathname(psearch),
6685                                            "reprobe");
6686                                        child = NULL;
6687                                        sdchild = NULL;
6688                                }

6690                        } else {
6691                                SCSI_HBA_LOG((_LOG(2), NULL, child,
6692                                    "%s no reprobe",
6693                                    mdi_pi_spathname(psearch)));

6695                                child = NULL;
6696                                sdchild = NULL;
6697                        }
6698                }
6699        }

6701        /* If asked for path_instance, return it. */
6702        if (ppi)
6703                *ppi = pi;

6705        return (sdchild);
6706 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26830 Wed Aug 19 07:25:14 2015
new/usr/src/uts/common/io/scsi/impl/scsi_watch.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 600 /*
 601  * The routines scsi_watch_thread & scsi_watch_request_intr are
 602  * on different threads.
 603  * If there is no work to be done by the lower level driver
 604  * then swr->swr_busy will not be set.
 605  * In this case we will call CALLB_CPR_SAFE_BEGIN before
 606  * calling cv_timedwait.
 607  * In the other case where there is work to be done by
 608  * the lower level driver then the flag swr->swr_busy will
 609  * be set.
 610  * We cannot call CALLB_CPR_SAFE_BEGIN at this point the reason
 611  * is the intr thread can interfere with our operations. So
 612  * we do a cv_timedwait here. Now at the completion of the
 613  * lower level driver's work we will call CALLB_CPR_SAFE_BEGIN
 614  * in scsi_watch_request_intr.
 615  * In all the cases we will call CALLB_CPR_SAFE_END only if
 616  * we already called a CALLB_CPR_SAFE_BEGIN and this is flagged
 617  * by sw_cpr_flag.
 618  * Warlock has a problem when we use different locks
 619  * on the same type of structure in different contexts.
 620  * We use callb_cpr_t in both scsi_watch and esp_callback threads.
 621  * we use different mutexe's in different threads. And
 622  * this is not acceptable to warlock. To avoid this
 623  * problem we use the same name for the mutex in
 624  * both scsi_watch & esp_callback. when __lock_lint is not defined
 625  * esp_callback uses the mutex on the stack and in scsi_watch
 626  * a static variable. But when __lock_lint is defined
 627  * we make a mutex which is global in esp_callback and
 628  * a external mutex for scsi_watch.
 629  */
 630 static int sw_cmd_count = 0;
 631 static int sw_cpr_flag = 0;
 632 static callb_cpr_t cpr_info;
 633 #ifndef __lock_lint
 634 static kmutex_t cpr_mutex;
 635 #else
 636 extern kmutex_t cpr_mutex;
 637 #endif

 639 #if !defined(lint)
 640 _NOTE(MUTEX_PROTECTS_DATA(cpr_mutex, cpr_info))
 641 _NOTE(MUTEX_PROTECTS_DATA(cpr_mutex, sw_cmd_count))
 642 #endif
 643 /*
 644  * the scsi watch thread:
 645  * it either wakes up if there is work to do or if the cv_timeait
 646  * timed out
 647  * normally, it wakes up every <delay> seconds and checks the list.
 648  * the interval is not very accurate if the cv was signalled but that
 649  * really doesn't matter much
 650  * it is more important that we fire off all TURs simulataneously so
 651  * we don't have to wake up frequently
 652  */
 653 static void
 654 scsi_watch_thread()
 655 {
 656         struct scsi_watch_request       *swr, *next;
 657         clock_t                         last_delay = 0;
```



```
 658         clock_t                         next_delay = 0;
 659         clock_t                         onesec = drv_sectohz(1);
 659         clock_t                         onesec = drv_usectohz(1000000);
 660         clock_t                         exit_delay = 60 * onesec;

 662         SW_DEBUG((dev_info_t *)NULL, sw_label, SCSI_DEBUG,
 663             "scsi_watch_thread: Entering ...\n");

 665 #if !defined(lint)
 666         _NOTE(NO_COMPETING_THREADS_NOW);
 667 #endif
 668         mutex_init(&cpr_mutex, NULL, MUTEX_DRIVER, NULL);
 669         CALLB_CPR_INIT(&cpr_info,
 670             &cpr_mutex, callb_generic_cpr, "scsi_watch");
 671         sw_cpr_flag = 0;
 672 #if !defined(lint)
 673         /*LINTED*/
 674         _NOTE(COMPETING_THREADS_NOW);
 675 #endif
 676         /*
 677          * grab the mutex and wait for work
 678          */
 679         mutex_enter(&sw.sw_mutex);
 680         if (sw.sw_head == NULL) {
 681                 cv_wait(&sw.sw_cv, &sw.sw_mutex);
 682         }

 684         /*
 685          * now loop forever for work; if queue is empty exit
 686          */
 687         for (;;) {
 688 head:
 689                 swr = sw.sw_head;
 690                 while (swr) {

 692                         /*
 693                          * If state is not running, wait for scsi_watch_resume
 694                          * to signal restart, but before going into cv_wait
 695                          * need to let the PM framework know that it is safe
 696                          * to stop this thread for CPR
 697                          */
 698                         if (sw.sw_state != SW_RUNNING) {
 699                                 SW_DEBUG(0, sw_label, SCSI_DEBUG,
 700                                     "scsi_watch_thread suspended\n");
 701                                 mutex_enter(&cpr_mutex);
 702                                 if (!sw_cmd_count) {
 703                                         CALLB_CPR_SAFE_BEGIN(&cpr_info);
 704                                         sw_cpr_flag = 1;
 705                                 }
 706                                 mutex_exit(&cpr_mutex);
 707                                 sw.swr_current = swr;
 708                                 cv_wait(&sw.sw_cv, &sw.sw_mutex);


 711                                 /*
 712                                  * Need to let the PM framework know that it
 713                                  * is no longer safe to stop the thread for
 714                                  * CPR.
 715                                  */
 716                                 mutex_exit(&sw.sw_mutex);
 717                                 mutex_enter(&cpr_mutex);
 718                                 if (sw_cpr_flag == 1) {
 719                                         CALLB_CPR_SAFE_END(
 720                                             &cpr_info, &cpr_mutex);
 721                                         sw_cpr_flag = 0;
 722                                 }
```

```
723                                     mutex_exit(&cpr_mutex);
724                                     mutex_enter(&sw.sw_mutex);
725                                     if (SUSPEND_DESTROY == swr->suspend_destroy) {
726                                             cv_destroy(&swr->swr_terminate_cv);
727                                             kmem_free((caddr_t)swr,
728                                                 sizeof (struct scsi_watch_request));
729                                             goto head;
730                                     } else {
731                                             sw.swr_current = NULL;
732                                     }
733                             }
734                             if (next_delay == 0) {
735                                     next_delay = swr->swr_timeout;
736                             } else {
737                                     next_delay = min(swr->swr_timeout, next_delay);
738                             }
739
740                             swr->swr_timeout -= last_delay;
741                             next = swr->swr_next;
742
743                             SW_DEBUG((dev_info_t *)NULL, sw_label, SCSI_DEBUG,
744                                 "scsi_watch_thread: "
745                                 "swr(0x%p),what=%x,timeout=%lx,"
746                                 "interval=%lx,delay=%lx\n",
747                                 (void *)swr, swr->swr_what, swr->swr_timeout,
748                                 swr->swr_interval, last_delay);
749
750                             switch (swr->swr_what) {
751                             case SWR_SUSPENDED:
752                             case SWR_SUSPEND_REQUESTED:
753                                     /* if we are suspended, don't do anything */
754                                     break;
755
756                             case SWR_STOP:
757                                     if (swr->swr_busy == 0) {
758                                             scsi_watch_request_destroy(swr);
759                                     }
760                                     break;
761
762                             default:
763                                     if (swr->swr_timeout <= 0 && !swr->swr_busy) {
764                                             swr->swr_busy = 1;
765                                             swr->swr_timeout = swr->swr_interval;
766
767                                             /*
768                                              * submit the cmd and let the completion
769                                              * function handle the result
770                                              * release the mutex (good practice)
771                                              * this should be safe even if the list
772                                              * is changing
773                                              */
774                                             mutex_exit(&sw.sw_mutex);
775                                             mutex_enter(&cpr_mutex);
776                                             sw_cmd_count++;
777                                             mutex_exit(&cpr_mutex);
778                                             SW_DEBUG((dev_info_t *)NULL,
779                                                 sw_label, SCSI_DEBUG,
780                                                 "scsi_watch_thread: "
781                                                 "Starting TUR\n");
782                                             if (scsi_transport(swr->swr_pkt) !=
783                                                 TRAN_ACCEPT) {
784
785                                                     /*
786                                                      * try again later
787                                                      */
788                                                     swr->swr_busy = 0;
```



```
789                                                     SW_DEBUG((dev_info_t *)NULL,
790                                                         sw_label, SCSI_DEBUG,
791                                                         "scsi_watch_thread: "
792                                                         "Transport Failed\n");
793                                                     mutex_enter(&cpr_mutex);
794                                                     sw_cmd_count--;
795                                                     mutex_exit(&cpr_mutex);
796                                             }
797                                             mutex_enter(&sw.sw_mutex);
798                                     }
799                                     break;
800                             }
801                             swr = next;
802                             if (sw.sw_flags & SW_START_HEAD) {
803                                     sw.sw_flags &= ~SW_START_HEAD;
804                                     goto head;
805                             }
806                     }
807
808                     /*
809                      * delay using cv_timedwait; we return when
810                      * signalled or timed out
811                      */
812                     if (sw.sw_head != NULL) {
813                             if (next_delay <= 0) {
814                                     next_delay = onesec;
815                             }
816                     } else {
817                             next_delay = exit_delay;
818                     }
819
820                     mutex_enter(&cpr_mutex);
821                     if (!sw_cmd_count) {
822                             CALLB_CPR_SAFE_BEGIN(&cpr_info);
823                             sw_cpr_flag = 1;
824                     }
825                     mutex_exit(&cpr_mutex);
826                     /*
827                      * if we return from cv_timedwait because we were
828                      * signalled, the delay is not accurate but that doesn't
829                      * really matter
830                      */
831                     (void) cv_reltimedwait(&sw.sw_cv, &sw.sw_mutex, next_delay,
832                         TR_CLOCK_TICK);
833                     mutex_exit(&sw.sw_mutex);
834                     mutex_enter(&cpr_mutex);
835                     if (sw_cpr_flag == 1) {
836                             CALLB_CPR_SAFE_END(&cpr_info, &cpr_mutex);
837                             sw_cpr_flag = 0;
838                     }
839                     mutex_exit(&cpr_mutex);
840                     mutex_enter(&sw.sw_mutex);
841                     last_delay = next_delay;
842                     next_delay = 0;
843
844                     /*
845                      * is there still work to do?
846                      */
847                     if (sw.sw_head == NULL) {
848                             break;
849                     }
850             }
851
852             /*
853              * no more work to do, reset sw_thread and exit
854              */
```

```
 855         sw.sw_thread = 0;
 856         mutex_exit(&sw.sw_mutex);
 857 #ifndef __lock_lint
 858         mutex_enter(&cpr_mutex);
 859         CALLB_CPR_EXIT(&cpr_info);
 860 #endif
 861         mutex_destroy(&cpr_mutex);
 862         SW_DEBUG((dev_info_t *)NULL, sw_label, SCSI_DEBUG,
 863             "scsi_watch_thread: Exiting ...\n");
 864 }
```

unchanged_portion_omitted_

```
**********************************************************
  911439 Wed Aug 19 07:25:14 2015
new/usr/src/uts/common/io/scsi/targets/sd.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


6142 /*
6143  *    Function: sd_ddi_suspend
6144  *
6145  * Description: Performs system power-down operations. This includes
6146  *              setting the drive state to indicate its suspended so
6147  *              that no new commands will be accepted. Also, wait for
6148  *              all commands that are in transport or queued to a timer
6149  *              for retry to complete. All timeout threads are cancelled.
6150  *
6151  * Return Code: DDI_FAILURE or DDI_SUCCESS
6152  *
6153  *     Context: Kernel thread context
6154  */

6156 static int
6157 sd_ddi_suspend(dev_info_t *devi)
6158 {
6159         struct  sd_lun  *un;
6160         clock_t         wait_cmds_complete;

6162         un = ddi_get_soft_state(sd_state, ddi_get_instance(devi));
6163         if (un == NULL) {
6164                 return (DDI_FAILURE);
6165         }

6167         SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: entry\n");

6169         mutex_enter(SD_MUTEX(un));

6171         /* Return success if the device is already suspended. */
6172         if (un->un_state == SD_STATE_SUSPENDED) {
6173                 mutex_exit(SD_MUTEX(un));
6174                 SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: "
6175                     "device already suspended, exiting\n");
6176                 return (DDI_SUCCESS);
6177         }

6179         /* Return failure if the device is being used by HA */
6180         if (un->un_resvd_status &
6181             (SD_RESERVE | SD_WANT_RESERVE | SD_LOST_RESERVE)) {
6182                 mutex_exit(SD_MUTEX(un));
6183                 SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: "
6184                     "device in use by HA, exiting\n");
6185                 return (DDI_FAILURE);
6186         }

6188         /*
6189          * Return failure if the device is in a resource wait
6190          * or power changing state.
6191          */
6192         if ((un->un_state == SD_STATE_RWAIT) ||
6193             (un->un_state == SD_STATE_PM_CHANGING)) {
6194                 mutex_exit(SD_MUTEX(un));
6195                 SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: "
6196                     "device in resource wait state, exiting\n");
6197                 return (DDI_FAILURE);
6198         }
```



```
6201         un->un_save_state = un->un_last_state;
6202         New_state(un, SD_STATE_SUSPENDED);

6204         /*
6205          * Wait for all commands that are in transport or queued to a timer
6206          * for retry to complete.
6207          *
6208          * While waiting, no new commands will be accepted or sent because of
6209          * the new state we set above.
6210          *
6211          * Wait till current operation has completed. If we are in the resource
6212          * wait state (with an intr outstanding) then we need to wait till the
6213          * intr completes and starts the next cmd. We want to wait for
6214          * SD_WAIT_CMDS_COMPLETE seconds before failing the DDI_SUSPEND.
6215          */
6216         wait_cmds_complete = ddi_get_lbolt() +
6217             drv_sectohz(sd_wait_cmds_complete);
6217             (sd_wait_cmds_complete * drv_usectohz(1000000));

6219         while (un->un_ncmds_in_transport != 0) {
6220                 /*
6221                  * Fail if commands do not finish in the specified time.
6222                  */
6223                 if (cv_timedwait(&un->un_disk_busy_cv, SD_MUTEX(un),
6224                     wait_cmds_complete) == -1) {
6225                         /*
6226                          * Undo the state changes made above. Everything
6227                          * must go back to it's original value.
6228                          */
6229                         Restore_state(un);
6230                         un->un_last_state = un->un_save_state;
6231                         /* Wake up any threads that might be waiting. */
6232                         cv_broadcast(&un->un_suspend_cv);
6233                         mutex_exit(SD_MUTEX(un));
6234                         SD_ERROR(SD_LOG_IO_PM, un,
6235                             "sd_ddi_suspend: failed due to outstanding cmds\n");
6236                         SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: exiting\n");
6237                         return (DDI_FAILURE);
6238                 }
6239         }

6241         /*
6242          * Cancel SCSI watch thread and timeouts, if any are active
6243          */

6245         if (SD_OK_TO_SUSPEND_SCSI_WATCHER(un)) {
6246                 opaque_t temp_token = un->un_swr_token;
6247                 mutex_exit(SD_MUTEX(un));
6248                 scsi_watch_suspend(temp_token);
6249                 mutex_enter(SD_MUTEX(un));
6250         }

6252         if (un->un_reset_throttle_timeid != NULL) {
6253                 timeout_id_t temp_id = un->un_reset_throttle_timeid;
6254                 un->un_reset_throttle_timeid = NULL;
6255                 mutex_exit(SD_MUTEX(un));
6256                 (void) untimeout(temp_id);
6257                 mutex_enter(SD_MUTEX(un));
6258         }

6260         if (un->un_dcvb_timeid != NULL) {
6261                 timeout_id_t temp_id = un->un_dcvb_timeid;
6262                 un->un_dcvb_timeid = NULL;
6263                 mutex_exit(SD_MUTEX(un));
6264                 (void) untimeout(temp_id);
```

```
6265                    mutex_enter(SD_MUTEX(un));
6266            }

6268            mutex_enter(&un->un_pm_mutex);
6269            if (un->un_pm_timeid != NULL) {
6270                    timeout_id_t temp_id = un->un_pm_timeid;
6271                    un->un_pm_timeid = NULL;
6272                    mutex_exit(&un->un_pm_mutex);
6273                    mutex_exit(SD_MUTEX(un));
6274                    (void) untimeout(temp_id);
6275                    mutex_enter(SD_MUTEX(un));
6276            } else {
6277                    mutex_exit(&un->un_pm_mutex);
6278            }

6280            if (un->un_rmw_msg_timeid != NULL) {
6281                    timeout_id_t temp_id = un->un_rmw_msg_timeid;
6282                    un->un_rmw_msg_timeid = NULL;
6283                    mutex_exit(SD_MUTEX(un));
6284                    (void) untimeout(temp_id);
6285                    mutex_enter(SD_MUTEX(un));
6286            }

6288            if (un->un_retry_timeid != NULL) {
6289                    timeout_id_t temp_id = un->un_retry_timeid;
6290                    un->un_retry_timeid = NULL;
6291                    mutex_exit(SD_MUTEX(un));
6292                    (void) untimeout(temp_id);
6293                    mutex_enter(SD_MUTEX(un));

6295                    if (un->un_retry_bp != NULL) {
6296                            un->un_retry_bp->av_forw = un->un_waitq_headp;
6297                            un->un_waitq_headp = un->un_retry_bp;
6298                            if (un->un_waitq_tailp == NULL) {
6299                                    un->un_waitq_tailp = un->un_retry_bp;
6300                            }
6301                            un->un_retry_bp = NULL;
6302                            un->un_retry_statp = NULL;
6303                    }
6304            }

6306            if (un->un_direct_priority_timeid != NULL) {
6307                    timeout_id_t temp_id = un->un_direct_priority_timeid;
6308                    un->un_direct_priority_timeid = NULL;
6309                    mutex_exit(SD_MUTEX(un));
6310                    (void) untimeout(temp_id);
6311                    mutex_enter(SD_MUTEX(un));
6312            }

6314            if (un->un_f_is_fibre == TRUE) {
6315                    /*
6316                     * Remove callbacks for insert and remove events
6317                     */
6318                    if (un->un_insert_event != NULL) {
6319                            mutex_exit(SD_MUTEX(un));
6320                            (void) ddi_remove_event_handler(un->un_insert_cb_id);
6321                            mutex_enter(SD_MUTEX(un));
6322                            un->un_insert_event = NULL;
6323                    }

6325                    if (un->un_remove_event != NULL) {
6326                            mutex_exit(SD_MUTEX(un));
6327                            (void) ddi_remove_event_handler(un->un_remove_cb_id);
6328                            mutex_enter(SD_MUTEX(un));
6329                            un->un_remove_event = NULL;
6330                    }
```



```
6331            }

6333            mutex_exit(SD_MUTEX(un));

6335            SD_TRACE(SD_LOG_IO_PM, un, "sd_ddi_suspend: exit\n");

6337            return (DDI_SUCCESS);
6338    }
```

*_____unchanged_portion_omitted_*

```
6608    /*
6609     *    Function: sdpower
6610     *
6611     * Description: PM entry point.
6612     *
6613     * Return Code: DDI_SUCCESS
6614     *              DDI_FAILURE
6615     *
6616     *     Context: Kernel thread context
6617     */

6619    static int
6620    sdpower(dev_info_t *devi, int component, int level)
6621    {
6622            struct sd_lun   *un;
6623            int             instance;
6624            int             rval = DDI_SUCCESS;
6625            uint_t          i, log_page_size, maxcycles, ncycles;
6626            uchar_t         *log_page_data;
6627            int             log_sense_page;
6628            int             medium_present;
6629            time_t          intvlp;
6630            struct pm_trans_data    sd_pm_tran_data;
6631            uchar_t         save_state;
6632            int             sval;
6633            uchar_t         state_before_pm;
6634            int             got_semaphore_here;
6635            sd_ssc_t        *ssc;
6636            int     last_power_level;

6638            instance = ddi_get_instance(devi);

6640            if (((un = ddi_get_soft_state(sd_state, instance)) == NULL) ||
6641                !SD_PM_IS_LEVEL_VALID(un, level) || component != 0) {
6642                    return (DDI_FAILURE);
6643            }

6645            ssc = sd_ssc_init(un);

6647            SD_TRACE(SD_LOG_IO_PM, un, "sdpower: entry, level = %d\n", level);

6649            /*
6650             * Must synchronize power down with close.
6651             * Attempt to decrement/acquire the open/close semaphore,
6652             * but do NOT wait on it. If it's not greater than zero,
6653             * ie. it can't be decremented without waiting, then
6654             * someone else, either open or close, already has it
6655             * and the try returns 0. Use that knowledge here to determine
6656             * if it's OK to change the device power level.
6657             * Also, only increment it on exit if it was decremented, ie. gotten,
6658             * here.
6659             */
6660            got_semaphore_here = sema_tryp(&un->un_semoclose);

6662            mutex_enter(SD_MUTEX(un));
```

```
6664         SD_INFO(SD_LOG_POWER, un, "sdpower: un_ncmds_in_driver = %ld\n",
6665             un->un_ncmds_in_driver);

6667         /*
6668          * If un_ncmds_in_driver is non-zero it indicates commands are
6669          * already being processed in the driver, or if the semaphore was
6670          * not gotten here it indicates an open or close is being processed.
6671          * At the same time somebody is requesting to go to a lower power
6672          * that can't perform I/O, which can't happen, therefore we need to
6673          * return failure.
6674          */
6675         if ((!SD_PM_IS_IO_CAPABLE(un, level)) &&
6676             ((un->un_ncmds_in_driver != 0) || (got_semaphore_here == 0))) {
6677                 mutex_exit(SD_MUTEX(un));

6679                 if (got_semaphore_here != 0) {
6680                         sema_v(&un->un_semoclose);
6681                 }
6682                 SD_TRACE(SD_LOG_IO_PM, un,
6683                     "sdpower: exit, device has queued cmds.\n");

6685                 goto sdpower_failed;
6686         }

6688         /*
6689          * if it is OFFLINE that means the disk is completely dead
6690          * in our case we have to put the disk in on or off by sending commands
6691          * Of course that will fail anyway so return back here.
6692          *
6693          * Power changes to a device that's OFFLINE or SUSPENDED
6694          * are not allowed.
6695          */
6696         if ((un->un_state == SD_STATE_OFFLINE) ||
6697             (un->un_state == SD_STATE_SUSPENDED)) {
6698                 mutex_exit(SD_MUTEX(un));

6700                 if (got_semaphore_here != 0) {
6701                         sema_v(&un->un_semoclose);
6702                 }
6703                 SD_TRACE(SD_LOG_IO_PM, un,
6704                     "sdpower: exit, device is off-line.\n");

6706                 goto sdpower_failed;
6707         }

6709         /*
6710          * Change the device's state to indicate it's power level
6711          * is being changed. Do this to prevent a power off in the
6712          * middle of commands, which is especially bad on devices
6713          * that are really powered off instead of just spun down.
6714          */
6715         state_before_pm = un->un_state;
6716         un->un_state = SD_STATE_PM_CHANGING;

6718         mutex_exit(SD_MUTEX(un));

6720         /*
6721          * If log sense command is not supported, bypass the
6722          * following checking, otherwise, check the log sense
6723          * information for this device.
6724          */
6725         if (SD_PM_STOP_MOTOR_NEEDED(un, level) &&
6726             un->un_f_log_sense_supported) {
6727                 /*
6728                  * Get the log sense information to understand whether the
```



```
6729                  * the powercycle counts have gone beyond the threshhold.
6730                  */
6731                 log_page_size = START_STOP_CYCLE_COUNTER_PAGE_SIZE;
6732                 log_page_data = kmem_zalloc(log_page_size, KM_SLEEP);

6734                 mutex_enter(SD_MUTEX(un));
6735                 log_sense_page = un->un_start_stop_cycle_page;
6736                 mutex_exit(SD_MUTEX(un));

6738                 rval = sd_send_scsi_LOG_SENSE(ssc, log_page_data,
6739                     log_page_size, log_sense_page, 0x01, 0, SD_PATH_DIRECT);

6741                 if (rval != 0) {
6742                         if (rval == EIO)
6743                                 sd_ssc_assessment(ssc, SD_FMT_STATUS_CHECK);
6744                         else
6745                                 sd_ssc_assessment(ssc, SD_FMT_IGNORE);
6746                 }

6748 #ifdef  SDDEBUG
6749                 if (sd_force_pm_supported) {
6750                         /* Force a successful result */
6751                         rval = 0;
6752                 }
6753 #endif
6754                 if (rval != 0) {
6755                         scsi_log(SD_DEVINFO(un), sd_label, CE_WARN,
6756                             "Log Sense Failed\n");

6758                         kmem_free(log_page_data, log_page_size);
6759                         /* Cannot support power management on those drives */

6761                         if (got_semaphore_here != 0) {
6762                                 sema_v(&un->un_semoclose);
6763                         }
6764                         /*
6765                          * On exit put the state back to it's original value
6766                          * and broadcast to anyone waiting for the power
6767                          * change completion.
6768                          */
6769                         mutex_enter(SD_MUTEX(un));
6770                         un->un_state = state_before_pm;
6771                         cv_broadcast(&un->un_suspend_cv);
6772                         mutex_exit(SD_MUTEX(un));
6773                         SD_TRACE(SD_LOG_IO_PM, un,
6774                             "sdpower: exit, Log Sense Failed.\n");

6776                         goto sdpower_failed;
6777                 }

6779                 /*
6780                  * From the page data - Convert the essential information to
6781                  * pm_trans_data
6782                  */
6783                 maxcycles =
6784                     (log_page_data[0x1c] << 24) | (log_page_data[0x1d] << 16) |
6785                     (log_page_data[0x1E] << 8)  | log_page_data[0x1F];

6787                 ncycles =
6788                     (log_page_data[0x24] << 24) | (log_page_data[0x25] << 16) |
6789                     (log_page_data[0x26] << 8)  | log_page_data[0x27];

6791                 if (un->un_f_pm_log_sense_smart) {
6792                         sd_pm_tran_data.un.smart_count.allowed = maxcycles;
6793                         sd_pm_tran_data.un.smart_count.consumed = ncycles;
6794                         sd_pm_tran_data.un.smart_count.flag = 0;
```

```
6795                        sd_pm_tran_data.format = DC_SMART_FORMAT;
6796                } else {
6797                        sd_pm_tran_data.un.scsi_cycles.lifemax = maxcycles;
6798                        sd_pm_tran_data.un.scsi_cycles.ncycles = ncycles;
6799                        for (i = 0; i < DC_SCSI_MFR_LEN; i++) {
6800                                sd_pm_tran_data.un.scsi_cycles.svc_date[i] =
6801                                    log_page_data[8+i];
6802                        }
6803                        sd_pm_tran_data.un.scsi_cycles.flag = 0;
6804                        sd_pm_tran_data.format = DC_SCSI_FORMAT;
6805                }

6807                kmem_free(log_page_data, log_page_size);

6809                /*
6810                 * Call pm_trans_check routine to get the Ok from
6811                 * the global policy
6812                 */
6813                rval = pm_trans_check(&sd_pm_tran_data, &intvlp);
6814 #ifdef  SDDEBUG
6815                if (sd_force_pm_supported) {
6816                        /* Force a successful result */
6817                        rval = 1;
6818                }
6819 #endif
6820                switch (rval) {
6821                case 0:
6822                        /*
6823                         * Not Ok to Power cycle or error in parameters passed
6824                         * Would have given the advised time to consider power
6825                         * cycle. Based on the new intvlp parameter we are
6826                         * supposed to pretend we are busy so that pm framework
6827                         * will never call our power entry point. Because of
6828                         * that install a timeout handler and wait for the
6829                         * recommended time to elapse so that power management
6830                         * can be effective again.
6831                         *
6832                         * To effect this behavior, call pm_busy_component to
6833                         * indicate to the framework this device is busy.
6834                         * By not adjusting un_pm_count the rest of PM in
6835                         * the driver will function normally, and independent
6836                         * of this but because the framework is told the device
6837                         * is busy it won't attempt powering down until it gets
6838                         * a matching idle. The timeout handler sends this.
6839                         * Note: sd_pm_entry can't be called here to do this
6840                         * because sdpower may have been called as a result
6841                         * of a call to pm_raise_power from within sd_pm_entry.
6842                         *
6843                         * If a timeout handler is already active then
6844                         * don't install another.
6845                         */
6846                        mutex_enter(&un->un_pm_mutex);
6847                        if (un->un_pm_timeid == NULL) {
6848                                un->un_pm_timeid =
6849                                    timeout(sd_pm_timeout_handler,
6850                                    un, drv_sectohz(intvlp));
6850                                    un, intvlp * drv_usectohz(1000000));
6851                                mutex_exit(&un->un_pm_mutex);
6852                                (void) pm_busy_component(SD_DEVINFO(un), 0);
6853                        } else {
6854                                mutex_exit(&un->un_pm_mutex);
6855                        }
6856                        if (got_semaphore_here != 0) {
6857                                sema_v(&un->un_semoclose);
6858                        }
6859                        /*
```



```
6860                         * On exit put the state back to it's original value
6861                         * and broadcast to anyone waiting for the power
6862                         * change completion.
6863                         */
6864                        mutex_enter(SD_MUTEX(un));
6865                        un->un_state = state_before_pm;
6866                        cv_broadcast(&un->un_suspend_cv);
6867                        mutex_exit(SD_MUTEX(un));

6869                        SD_TRACE(SD_LOG_IO_PM, un, "sdpower: exit, "
6870                            "trans check Failed, not ok to power cycle.\n");

6872                        goto sdpower_failed;
6873                case -1:
6874                        if (got_semaphore_here != 0) {
6875                                sema_v(&un->un_semoclose);
6876                        }
6877                        /*
6878                         * On exit put the state back to it's original value
6879                         * and broadcast to anyone waiting for the power
6880                         * change completion.
6881                         */
6882                        mutex_enter(SD_MUTEX(un));
6883                        un->un_state = state_before_pm;
6884                        cv_broadcast(&un->un_suspend_cv);
6885                        mutex_exit(SD_MUTEX(un));
6886                        SD_TRACE(SD_LOG_IO_PM, un,
6887                            "sdpower: exit, trans check command Failed.\n");

6889                        goto sdpower_failed;
6890                }
6891        }

6893        if (!SD_PM_IS_IO_CAPABLE(un, level)) {
6894                /*
6895                 * Save the last state... if the STOP FAILS we need it
6896                 * for restoring
6897                 */
6898                mutex_enter(SD_MUTEX(un));
6899                save_state = un->un_last_state;
6900                last_power_level = un->un_power_level;
6901                /*
6902                 * There must not be any cmds. getting processed
6903                 * in the driver when we get here. Power to the
6904                 * device is potentially going off.
6905                 */
6906                ASSERT(un->un_ncmds_in_driver == 0);
6907                mutex_exit(SD_MUTEX(un));

6909                /*
6910                 * For now PM suspend the device completely before spindle is
6911                 * turned off
6912                 */
6913                if ((rval = sd_pm_state_change(un, level, SD_PM_STATE_CHANGE))
6914                    == DDI_FAILURE) {
6915                        if (got_semaphore_here != 0) {
6916                                sema_v(&un->un_semoclose);
6917                        }
6918                        /*
6919                         * On exit put the state back to it's original value
6920                         * and broadcast to anyone waiting for the power
6921                         * change completion.
6922                         */
6923                        mutex_enter(SD_MUTEX(un));
6924                        un->un_state = state_before_pm;
6925                        un->un_power_level = last_power_level;
```

```
6926                        cv_broadcast(&un->un_suspend_cv);
6927                        mutex_exit(SD_MUTEX(un));
6928                        SD_TRACE(SD_LOG_IO_PM, un,
6929                            "sdpower: exit, PM suspend Failed.\n");
6930
6931                        goto sdpower_failed;
6932                }
6933        }
6934
6935        /*
6936         * The transition from SPINDLE_OFF to SPINDLE_ON can happen in open,
6937         * close, or strategy. Dump no long uses this routine, it uses it's
6938         * own code so it can be done in polled mode.
6939         */
6940
6941        medium_present = TRUE;
6942
6943        /*
6944         * When powering up, issue a TUR in case the device is at unit
6945         * attention.  Don't do retries. Bypass the PM layer, otherwise
6946         * a deadlock on un_pm_busy_cv will occur.
6947         */
6948        if (SD_PM_IS_IO_CAPABLE(un, level)) {
6949                sval = sd_send_scsi_TEST_UNIT_READY(ssc,
6950                    SD_DONT_RETRY_TUR | SD_BYPASS_PM);
6951                if (sval != 0)
6952                        sd_ssc_assessment(ssc, SD_FMT_IGNORE);
6953        }
6954
6955        if (un->un_f_power_condition_supported) {
6956                char *pm_condition_name[] = {"STOPPED", "STANDBY",
6957                    "IDLE", "ACTIVE"};
6958                SD_TRACE(SD_LOG_IO_PM, un,
6959                    "sdpower: sending \'%s\' power condition",
6960                    pm_condition_name[level]);
6961                sval = sd_send_scsi_START_STOP_UNIT(ssc, SD_POWER_CONDITION,
6962                    sd_pl2pc[level], SD_PATH_DIRECT);
6963        } else {
6964                SD_TRACE(SD_LOG_IO_PM, un, "sdpower: sending \'%s\' unit\n",
6965                    ((level == SD_SPINDLE_ON) ? "START" : "STOP"));
6966                sval = sd_send_scsi_START_STOP_UNIT(ssc, SD_START_STOP,
6967                    ((level == SD_SPINDLE_ON) ? SD_TARGET_START :
6968                    SD_TARGET_STOP), SD_PATH_DIRECT);
6969        }
6970        if (sval != 0) {
6971                if (sval == EIO)
6972                        sd_ssc_assessment(ssc, SD_FMT_STATUS_CHECK);
6973                else
6974                        sd_ssc_assessment(ssc, SD_FMT_IGNORE);
6975        }
6976
6977        /* Command failed, check for media present. */
6978        if ((sval == ENXIO) && un->un_f_has_removable_media) {
6979                medium_present = FALSE;
6980        }
6981
6982        /*
6983         * The conditions of interest here are:
6984         *   if a spindle off with media present fails,
6985         *      then restore the state and return an error.
6986         *   else if a spindle on fails,
6987         *      then return an error (there's no state to restore).
6988         * In all other cases we setup for the new state
6989         * and return success.
6990         */
6991        if (!SD_PM_IS_IO_CAPABLE(un, level)) {
```



```
6992                if ((medium_present == TRUE) && (sval != 0)) {
6993                        /* The stop command from above failed */
6994                        rval = DDI_FAILURE;
6995                        /*
6996                         * The stop command failed, and we have media
6997                         * present. Put the level back by calling the
6998                         * sd_pm_resume() and set the state back to
6999                         * it's previous value.
7000                         */
7001                        (void) sd_pm_state_change(un, last_power_level,
7002                            SD_PM_STATE_ROLLBACK);
7003                        mutex_enter(SD_MUTEX(un));
7004                        un->un_last_state = save_state;
7005                        mutex_exit(SD_MUTEX(un));
7006                } else if (un->un_f_monitor_media_state) {
7007                        /*
7008                         * The stop command from above succeeded.
7009                         * Terminate watch thread in case of removable media
7010                         * devices going into low power state. This is as per
7011                         * the requirements of pm framework, otherwise commands
7012                         * will be generated for the device (through watch
7013                         * thread), even when the device is in low power state.
7014                         */
7015                        mutex_enter(SD_MUTEX(un));
7016                        un->un_f_watcht_stopped = FALSE;
7017                        if (un->un_swr_token != NULL) {
7018                                opaque_t temp_token = un->un_swr_token;
7019                                un->un_f_watcht_stopped = TRUE;
7020                                un->un_swr_token = NULL;
7021                                mutex_exit(SD_MUTEX(un));
7022                                (void) scsi_watch_request_terminate(temp_token,
7023                                    SCSI_WATCH_TERMINATE_ALL_WAIT);
7024                        } else {
7025                                mutex_exit(SD_MUTEX(un));
7026                        }
7027                }
7028        } else {
7029                /*
7030                 * The level requested is I/O capable.
7031                 * Legacy behavior: return success on a failed spinup
7032                 * if there is no media in the drive.
7033                 * Do this by looking at medium_present here.
7034                 */
7035                if ((sval != 0) && medium_present) {
7036                        /* The start command from above failed */
7037                        rval = DDI_FAILURE;
7038                } else {
7039                        /*
7040                         * The start command from above succeeded
7041                         * PM resume the devices now that we have
7042                         * started the disks
7043                         */
7044                        (void) sd_pm_state_change(un, level,
7045                            SD_PM_STATE_CHANGE);
7046
7047                        /*
7048                         * Resume the watch thread since it was suspended
7049                         * when the device went into low power mode.
7050                         */
7051                        if (un->un_f_monitor_media_state) {
7052                                mutex_enter(SD_MUTEX(un));
7053                                if (un->un_f_watcht_stopped == TRUE) {
7054                                        opaque_t temp_token;
7055
7056                                        un->un_f_watcht_stopped = FALSE;
7057                                        mutex_exit(SD_MUTEX(un));
```

```
7058                                        temp_token =
7059                                            sd_watch_request_submit(un);
7060                                        mutex_enter(SD_MUTEX(un));
7061                                        un->un_swr_token = temp_token;
7062                                }
7063                                mutex_exit(SD_MUTEX(un));
7064                        }
7065                }
7066        }

7068        if (got_semaphore_here != 0) {
7069                sema_v(&un->un_semoclose);
7070        }
7071        /*
7072         * On exit put the state back to it's original value
7073         * and broadcast to anyone waiting for the power
7074         * change completion.
7075         */
7076        mutex_enter(SD_MUTEX(un));
7077        un->un_state = state_before_pm;
7078        cv_broadcast(&un->un_suspend_cv);
7079        mutex_exit(SD_MUTEX(un));

7081        SD_TRACE(SD_LOG_IO_PM, un, "sdpower: exit, status = 0x%x\n", rval);

7083        sd_ssc_fini(ssc);
7084        return (rval);

7086 sdpower_failed:

7088        sd_ssc_fini(ssc);
7089        return (DDI_FAILURE);
7090 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  476380 Wed Aug 19 07:25:15 2015
new/usr/src/uts/common/io/scsi/targets/st.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1203 /*
1204  * st_detach:
1205  *
1206  * we allow a detach if and only if:
1207  *      - no tape is currently inserted
1208  *      - tape position is at BOT or unknown
1209  *              (if it is not at BOT then a no rewind
1210  *              device was opened and we have to preserve state)
1211  *      - it must be in a closed state : no timeouts or scsi_watch requests
1212  *              will exist if it is closed, so we don't need to check for
1213  *              them here.
1214  */
1215 /*ARGSUSED*/
1216 static int
1217 st_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
1218 {
1219         int instance;
1220         int result;
1221         struct scsi_device *devp;
1222         struct scsi_tape *un;
1223         clock_t wait_cmds_complete;

1225         ST_ENTR(devi, st_detach);

1227         instance = ddi_get_instance(devi);

1229         if (!(un = ddi_get_soft_state(st_state, instance))) {
1230                 return (DDI_FAILURE);
1231         }

1233         mutex_enter(ST_MUTEX);

1235         /*
1236          * Clear error entry stack
1237          */
1238         st_empty_error_stack(un);

1240         mutex_exit(ST_MUTEX);

1242         switch (cmd) {

1244         case DDI_DETACH:
1245                 /*
1246                  * Undo what we did in st_attach & st_doattach,
1247                  * freeing resources and removing things we installed.
1248                  * The system framework guarantees we are not active
1249                  * with this devinfo node in any other entry points at
1250                  * this time.
1251                  */

1253                 ST_DEBUG(ST_DEVINFO, st_label, SCSI_DEBUG,
1254                     "st_detach: instance=%x, un=%p\n", instance,
1255                     (void *)un);

1257                 if (((un->un_dp->options & ST_UNLOADABLE) == 0) ||
1258                     ((un->un_rsvd_status & ST_APPLICATION_RESERVATIONS) != 0) ||
1259                     (un->un_ncmds != 0) || (un->un_quef != NULL) ||
1260                     (un->un_state != ST_STATE_CLOSED)) {
1261                         /*
```



```
1262                          * we cannot unload some targets because the
1263                          * inquiry returns junk unless immediately
1264                          * after a reset
1265                          */
1266                         ST_DEBUG2(ST_DEVINFO, st_label, SCSI_DEBUG,
1267                             "cannot unload instance %x\n", instance);
1268                         un->un_unit_attention_flags |= 4;
1269                         return (DDI_FAILURE);
1270                 }

1272                 /*
1273                  * if the tape has been removed then we may unload;
1274                  * do a test unit ready and if it returns NOT READY
1275                  * then we assume that it is safe to unload.
1276                  * as a side effect, pmode may be set to invalid if the
1277                  * the test unit ready fails;
1278                  * also un_state may be set to non-closed, so reset it
1279                  */
1280                 if ((un->un_dev) &&             /* Been opened since attach */
1281                     ((un->un_pos.pmode == legacy) &&
1282                     (un->un_pos.fileno > 0) ||  /* Known position not rewound */
1283                     (un->un_pos.blkno != 0)) || /* Or within first file */
1284                     ((un->un_pos.pmode == logical) &&
1285                     (un->un_pos.lgclblkno > 0))) {
1286                         mutex_enter(ST_MUTEX);
1287                         /*
1288                          * Send Test Unit Ready in the hopes that if
1289                          * the drive is not in the state we think it is.
1290                          * And the state will be changed so it can be detached.
1291                          * If the command fails to reach the device and
1292                          * the drive was not rewound or unloaded we want
1293                          * to fail the detach till a user command fails
1294                          * where after the detach will succead.
1295                          */
1296                         result = st_cmd(un, SCMD_TEST_UNIT_READY, 0, SYNC_CMD);
1297                         /*
1298                          * After TUR un_state may be set to non-closed,
1299                          * so reset it back.
1300                          */
1301                         un->un_state = ST_STATE_CLOSED;
1302                         mutex_exit(ST_MUTEX);
1303                 }
1304                 ST_DEBUG(ST_DEVINFO, st_label, SCSI_DEBUG,
1305                     "un_status=%x, fileno=%x, blkno=%x\n",
1306                     un->un_status, un->un_pos.fileno, un->un_pos.blkno);

1308                 /*
1309                  * check again:
1310                  * if we are not at BOT then it is not safe to unload
1311                  */
1312                 if ((un->un_dev) &&             /* Been opened since attach */
1313                     (result != EACCES) &&       /* drive is use by somebody */
1314                     ((((un->un_pos.pmode == legacy) &&
1315                     (un->un_pos.fileno > 0) ||  /* Known position not rewound */
1316                     (un->un_pos.blkno != 0)) || /* Or within first file */
1317                     ((un->un_pos.pmode == logical) &&
1318                     (un->un_pos.lgclblkno > 0))) &&
1319                     ((un->un_state == ST_STATE_CLOSED) &&
1320                     (un->un_laststate == ST_STATE_CLOSING)))) {

1322                         ST_DEBUG(ST_DEVINFO, st_label, SCSI_DEBUG,
1323                             "cannot detach: pmode=%d fileno=0x%x, blkno=0x%x"
1324                             " lgclblkno=0x%"PRIx64"\n", un->un_pos.pmode,
1325                             un->un_pos.fileno, un->un_pos.blkno,
1326                             un->un_pos.lgclblkno);
1327                         un->un_unit_attention_flags |= 4;
```

```
1328                        return (DDI_FAILURE);
1329                }

1331                /*
1332                 * Just To make sure that we have released the
1333                 * tape unit .
1334                 */
1335                if (un->un_dev && (un->un_rsvd_status & ST_RESERVE) &&
1336                    !DEVI_IS_DEVICE_REMOVED(devi)) {
1337                        mutex_enter(ST_MUTEX);
1338                        (void) st_reserve_release(un, ST_RELEASE, st_uscsi_cmd);
1339                        mutex_exit(ST_MUTEX);
1340                }

1342                /*
1343                 * now remove other data structures allocated in st_doattach()
1344                 */
1345                ST_DEBUG(ST_DEVINFO, st_label, SCSI_DEBUG,
1346                    "destroying/freeing\n");

1348                (void) scsi_reset_notify(ROUTE, SCSI_RESET_CANCEL,
1349                    st_reset_notification, (caddr_t)un);
1350                cv_destroy(&un->un_clscv);
1351                cv_destroy(&un->un_sbuf_cv);
1352                cv_destroy(&un->un_queue_cv);
1353                cv_destroy(&un->un_suspend_cv);
1354                cv_destroy(&un->un_tape_busy_cv);
1355                cv_destroy(&un->un_recov_buf_cv);

1357                if (un->un_recov_taskq) {
1358                        ddi_taskq_destroy(un->un_recov_taskq);
1359                }

1361                if (un->un_hib_tid) {
1362                        (void) untimeout(un->un_hib_tid);
1363                        un->un_hib_tid = 0;
1364                }

1366                if (un->un_delay_tid) {
1367                        (void) untimeout(un->un_delay_tid);
1368                        un->un_delay_tid = 0;
1369                }
1370                cv_destroy(&un->un_state_cv);

1372 #ifdef  __x86
1373                cv_destroy(&un->un_contig_mem_cv);

1375                if (un->un_contig_mem_hdl != NULL) {
1376                        ddi_dma_free_handle(&un->un_contig_mem_hdl);
1377                }
1378 #endif
1379                if (un->un_sbufp) {
1380                        freerbuf(un->un_sbufp);
1381                }
1382                if (un->un_recov_buf) {
1383                        freerbuf(un->un_recov_buf);
1384                }
1385                if (un->un_uscsi_rqs_buf) {
1386                        kmem_free(un->un_uscsi_rqs_buf, SENSE_LENGTH);
1387                }
1388                if (un->un_mspl) {
1389                        i_ddi_mem_free((caddr_t)un->un_mspl, NULL);
1390                }
1391                if (un->un_rqs) {
1392                        scsi_destroy_pkt(un->un_rqs);
1393                        scsi_free_consistent_buf(un->un_rqs_bp);
```



```
1394                }
1395                if (un->un_mkr_pkt) {
1396                        scsi_destroy_pkt(un->un_mkr_pkt);
1397                }
1398                if (un->un_arq_enabled) {
1399                        (void) scsi_ifsetcap(ROUTE, "auto-rqsense", 0, 1);
1400                }
1401                if (un->un_dp_size) {
1402                        kmem_free(un->un_dp, un->un_dp_size);
1403                }
1404                if (un->un_stats) {
1405                        kstat_delete(un->un_stats);
1406                        un->un_stats = (kstat_t *)0;
1407                }
1408                if (un->un_errstats) {
1409                        kstat_delete(un->un_errstats);
1410                        un->un_errstats = (kstat_t *)0;
1411                }
1412                if (un->un_media_id_len) {
1413                        kmem_free(un->un_media_id, un->un_media_id_len);
1414                }
1415                devp = ST_SCSI_DEVP;
1416                ddi_soft_state_free(st_state, instance);
1417                devp->sd_private = NULL;
1418                devp->sd_sense = NULL;
1419                scsi_unprobe(devp);
1420                ddi_prop_remove_all(devi);
1421                ddi_remove_minor_node(devi, NULL);
1422                ST_DEBUG(0, st_label, SCSI_DEBUG, "st_detach done\n");
1423                return (DDI_SUCCESS);

1425        case DDI_SUSPEND:

1427                /*
1428                 * Suspend/Resume
1429                 *
1430                 * To process DDI_SUSPEND, we must do the following:
1431                 *
1432                 *  - check ddi_removing_power to see if power will be turned
1433                 *    off. if so, return DDI_FAILURE
1434                 *  - check if we are already suspended,
1435                 *    if so, return DDI_FAILURE
1436                 *  - check if device state is CLOSED,
1437                 *    if not, return DDI_FAILURE.
1438                 *  - wait until outstanding operations complete
1439                 *  - save tape state
1440                 *  - block new operations
1441                 *  - cancel pending timeouts
1442                 *
1443                 */

1445                if (ddi_removing_power(devi)) {
1446                        return (DDI_FAILURE);
1447                }

1449                if (un->un_dev == 0)
1450                        un->un_dev = MTMINOR(instance);

1452                mutex_enter(ST_MUTEX);

1454                /*
1455                 * Shouldn't already be suspended, if so return failure
1456                 */
1457                if (un->un_pwr_mgmt == ST_PWR_SUSPENDED) {
1458                        mutex_exit(ST_MUTEX);
1459                        return (DDI_FAILURE);
```

```
1460                }
1461                if (un->un_state != ST_STATE_CLOSED) {
1462                        mutex_exit(ST_MUTEX);
1463                        return (DDI_FAILURE);
1464                }

1466                /*
1467                 * Wait for all outstanding I/O's to complete
1468                 *
1469                 * we wait on both ncmds and the wait queue for times
1470                 * when we are flushing after persistent errors are
1471                 * flagged, which is when ncmds can be 0, and the
1472                 * queue can still have I/O's.  This way we preserve
1473                 * order of biodone's.
1474                 */
1475                wait_cmds_complete = ddi_get_lbolt();
1476                wait_cmds_complete +=
1477                    drv_sectohz(st_wait_cmds_complete);
1477                    st_wait_cmds_complete * drv_usectohz(1000000);
1478                while (un->un_ncmds || un->un_quef ||
1479                    (un->un_state == ST_STATE_RESOURCE_WAIT)) {

1481                        if (cv_timedwait(&un->un_tape_busy_cv, ST_MUTEX,
1482                            wait_cmds_complete) == -1) {
1483                                /*
1484                                 * Time expired then cancel the command
1485                                 */
1486                                if (st_reset(un, RESET_LUN) == 0) {
1487                                        if (un->un_last_throttle) {
1488                                                un->un_throttle =
1489                                                    un->un_last_throttle;
1490                                        }
1491                                        mutex_exit(ST_MUTEX);
1492                                        return (DDI_FAILURE);
1493                                } else {
1494                                        break;
1495                                }
1496                        }
1497                }

1499                /*
1500                 * DDI_SUSPEND says that the system "may" power down, we
1501                 * remember the file and block number before rewinding.
1502                 * we also need to save state before issuing
1503                 * any WRITE_FILE_MARK command.
1504                 */
1505                (void) st_update_block_pos(un, st_cmd, 0);
1506                COPY_POS(&un->un_suspend_pos, &un->un_pos);


1509                /*
1510                 * Issue a zero write file fmk command to tell the drive to
1511                 * flush any buffered tape marks
1512                 */
1513                (void) st_cmd(un, SCMD_WRITE_FILE_MARK, 0, SYNC_CMD);

1515                /*
1516                 * Because not all tape drives correctly implement buffer
1517                 * flushing with the zero write file fmk command, issue a
1518                 * synchronous rewind command to force data flushing.
1519                 * st_validate_tapemarks() will do a rewind during DDI_RESUME
1520                 * anyway.
1521                 */
1522                (void) st_cmd(un, SCMD_REWIND, 0, SYNC_CMD);

1524                /* stop any new operations */
```



```
1525                un->un_pwr_mgmt = ST_PWR_SUSPENDED;
1526                un->un_throttle = 0;

1528                /*
1529                 * cancel any outstanding timeouts
1530                 */
1531                if (un->un_delay_tid) {
1532                        timeout_id_t temp_id = un->un_delay_tid;
1533                        un->un_delay_tid = 0;
1534                        un->un_tids_at_suspend |= ST_DELAY_TID;
1535                        mutex_exit(ST_MUTEX);
1536                        (void) untimeout(temp_id);
1537                        mutex_enter(ST_MUTEX);
1538                }

1540                if (un->un_hib_tid) {
1541                        timeout_id_t temp_id = un->un_hib_tid;
1542                        un->un_hib_tid = 0;
1543                        un->un_tids_at_suspend |= ST_HIB_TID;
1544                        mutex_exit(ST_MUTEX);
1545                        (void) untimeout(temp_id);
1546                        mutex_enter(ST_MUTEX);
1547                }

1549                /*
1550                 * Suspend the scsi_watch_thread
1551                 */
1552                if (un->un_swr_token) {
1553                        opaque_t temp_token = un->un_swr_token;
1554                        mutex_exit(ST_MUTEX);
1555                        scsi_watch_suspend(temp_token);
1556                } else {
1557                        mutex_exit(ST_MUTEX);
1558                }

1560                return (DDI_SUCCESS);

1562        default:
1563                ST_DEBUG(0, st_label, SCSI_DEBUG, "st_detach failed\n");
1564                return (DDI_FAILURE);
1565        }
1566 }
______unchanged_portion_omitted_
6422 #endif


6425 /*
6426  * Command start && done functions
6427  */

6429 /*
6430  * st_start()
6431  *
6432  * Called from:
6433  *  st_strategy() to start a command.
6434  *  st_runout() to retry when scsi_pkt allocation fails on previous attempt(s).
6435  *  st_attach() when resuming from power down state.
6436  *  st_start_restart() to retry transport when device was previously busy.
6437  *  st_done_and_mutex_exit() to start the next command when previous is done.
6438  *
6439  * On entry:
6440  *  scsi_pkt may or may not be allocated.
6441  *
6442  */
6443 static void
6444 st_start(struct scsi_tape *un)
```

```
6445 {
6446         struct buf *bp;
6447         int status;
6448         int queued;

6450         ST_FUNC(ST_DEVINFO, st_start);
6451         ASSERT(mutex_owned(ST_MUTEX));

6453         ST_DEBUG3(ST_DEVINFO, st_label, SCSI_DEBUG,
6454             "st_start(): dev = 0x%lx\n", un->un_dev);

6456         if (un->un_recov_buf_busy) {
6457                 /* recovery commands can happen anytime */
6458                 bp = un->un_recov_buf;
6459                 queued = 0;
6460         } else if (un->un_sbuf_busy) {
6461                 /* sbuf commands should only happen with an empty queue. */
6462                 ASSERT(un->un_quef == NULL);
6463                 ASSERT(un->un_runqf == NULL);
6464                 bp = un->un_sbufp;
6465                 queued = 0;
6466         } else if (un->un_quef != NULL) {
6467                 if (un->un_persistence && un->un_persist_errors) {
6468                         return;
6469                 }
6470                 bp = un->un_quef;
6471                 queued = 1;
6472         } else {
6473                 scsi_log(ST_DEVINFO, st_label, SCSI_DEBUG,
6474                     "st_start() returning no buf found\n");
6475                 return;
6476         }

6478         ASSERT((bp->b_flags & B_DONE) == 0);

6480         /*
6481          * Don't send more than un_throttle commands to the HBA
6482          */
6483         if ((un->un_throttle <= 0) || (un->un_ncmds >= un->un_throttle)) {
6484                 /*
6485                  * if doing recovery we know there is outstanding commands.
6486                  */
6487                 if (bp != un->un_recov_buf) {
6488                         ST_DEBUG2(ST_DEVINFO, st_label, SCSI_DEBUG,
6489                             "st_start returning throttle = %d or ncmds = %d\n",
6490                             un->un_throttle, un->un_ncmds);
6491                         if (un->un_ncmds == 0) {
6492                                 typedef void (*func)();
6493                                 func fnc = (func)st_runout;

6495                                 scsi_log(ST_DEVINFO, st_label, SCSI_DEBUG,
6496                                     "Sending delayed start to st_runout()\n");
6497                                 mutex_exit(ST_MUTEX);
6498                                 (void) timeout(fnc, un, drv_sectohz(1));
6498                                 (void) timeout(fnc, un, drv_usectohz(1000000));
6499                                 mutex_enter(ST_MUTEX);
6500                         }
6501                         return;
6502                 }
6503         }

6505         /*
6506          * If the buf has no scsi_pkt call st_make_cmd() to get one and
6507          * build the command.
6508          */
6509         if (BP_PKT(bp) == NULL) {
```



```
6510                 ASSERT((bp->b_flags & B_DONE) == 0);
6511                 st_make_cmd(un, bp, st_runout);
6512                 ASSERT((bp->b_flags & B_DONE) == 0);
6513                 status = geterror(bp);

6515                 /*
6516                  * Some HBA's don't call bioerror() to set an error.
6517                  * And geterror() returns zero if B_ERROR is not set.
6518                  * So if we get zero we must check b_error.
6519                  */
6520                 if (status == 0 && bp->b_error != 0) {
6521                         status = bp->b_error;
6522                         bioerror(bp, status);
6523                 }

6525                 /*
6526                  * Some HBA's convert DDI_DMA_NORESOURCES into ENOMEM.
6527                  * In tape ENOMEM has special meaning so we'll change it.
6528                  */
6529                 if (status == ENOMEM) {
6530                         status = 0;
6531                         bioerror(bp, status);
6532                 }

6534                 /*
6535                  * Did it fail and is it retryable?
6536                  * If so return and wait for the callback through st_runout.
6537                  * Also looks like scsi_init_pkt() will setup a callback even
6538                  * if it isn't retryable.
6539                  */
6540                 if (BP_PKT(bp) == NULL) {
6541                         if (status == 0) {
6542                                 /*
6543                                  * If first attempt save state.
6544                                  */
6545                                 if (un->un_state != ST_STATE_RESOURCE_WAIT) {
6546                                         un->un_laststate = un->un_state;
6547                                         un->un_state = ST_STATE_RESOURCE_WAIT;
6548                                 }
6549                                 ST_DEBUG2(ST_DEVINFO, st_label, SCSI_DEBUG,
6550                                     "temp no resources for pkt\n");
6551                         } else if (status == EINVAL) {
6552                                 scsi_log(ST_DEVINFO, st_label, SCSI_DEBUG,
6553                                     "scsi_init_pkt rejected pkt as too big\n");
6554                                 if (un->un_persistence) {
6555                                         st_set_pe_flag(un);
6556                                 }
6557                         } else {
6558                                 /*
6559                                  * Unlikely that it would be retryable then not.
6560                                  */
6561                                 if (un->un_state == ST_STATE_RESOURCE_WAIT) {
6562                                         un->un_state = un->un_laststate;
6563                                 }
6564                                 scsi_log(ST_DEVINFO, st_label, SCSI_DEBUG,
6565                                     "perm no resources for pkt errno = 0x%x\n",
6566                                     status);
6567                         }
6568                         return;
6569                 }
6570                 /*
6571                  * Worked this time set the state back.
6572                  */
6573                 if (un->un_state == ST_STATE_RESOURCE_WAIT) {
6574                         un->un_state = un->un_laststate;
6575                 }
```

```
6576         }

6578         if (queued) {
6579                 /*
6580                  * move from waitq to runq
6581                  */
6582                 (void) st_remove_from_queue(&un->un_quef, &un->un_quel, bp);
6583                 st_add_to_queue(&un->un_runqf, &un->un_runql, un->un_runql, bp);
6584         }


6587 #ifdef STDEBUG
6588         st_start_dump(un, bp);
6589 #endif

6591         /* could not get here if throttle was zero */
6592         un->un_last_throttle = un->un_throttle;
6593         un->un_throttle = 0;    /* so nothing else will come in here */
6594         un->un_ncmds++;

6596         ST_DO_KSTATS(bp, kstat_waitq_to_runq);

6598         status = st_transport(un, BP_PKT(bp));

6600         if (un->un_last_throttle) {
6601                 un->un_throttle = un->un_last_throttle;
6602         }

6604         if (status != TRAN_ACCEPT) {
6605                 ST_DO_KSTATS(bp, kstat_runq_back_to_waitq);
6606                 ST_DEBUG(ST_DEVINFO, st_label, CE_WARN,
6607                     "Unhappy transport packet status 0x%x\n", status);

6609                 if (status == TRAN_BUSY) {
6610                         pkt_info *pkti = BP_PKT(bp)->pkt_private;

6612                         /*
6613                          * If command recovery is enabled and this isn't
6614                          * a recovery command try command recovery.
6615                          */
6616                         if (pkti->privatelen == sizeof (recov_info) &&
6617                             bp != un->un_recov_buf) {
6618                                 ST_RECOV(ST_DEVINFO, st_label, CE_WARN,
6619                                     "Command Recovery called on busy send\n");
6620                                 if (st_command_recovery(un, BP_PKT(bp),
6621                                     ATTEMPT_RETRY) == JUST_RETURN) {
6622                                         return;
6623                                 }
6624                         } else {
6625                                 mutex_exit(ST_MUTEX);
6626                                 if (st_handle_start_busy(un, bp,
6627                                     ST_TRAN_BUSY_TIMEOUT, queued) == 0) {
6628                                         mutex_enter(ST_MUTEX);
6629                                         return;
6630                                 }
6631                                 /*
6632                                  * if too many retries, fail the transport
6633                                  */
6634                                 mutex_enter(ST_MUTEX);
6635                         }
6636                 }
6637                 scsi_log(ST_DEVINFO, st_label, CE_WARN,
6638                     "transport rejected %d\n", status);
6639                 bp->b_resid = bp->b_bcount;

6641                 ST_DO_KSTATS(bp, kstat_waitq_exit);
```



```
6642                 ST_DO_ERRSTATS(un, st_transerrs);
6643                 if ((bp == un->un_recov_buf) && (status == TRAN_BUSY)) {
6644                         st_bioerror(bp, EBUSY);
6645                 } else {
6646                         st_bioerror(bp, EIO);
6647                         st_set_pe_flag(un);
6648                 }
6649                 st_done_and_mutex_exit(un, bp);
6650                 mutex_enter(ST_MUTEX);
6651         }

6653         ASSERT(mutex_owned(ST_MUTEX));
6654 }
______unchanged_portion_omitted_

17605 static int
17606 st_test_path_to_device(struct scsi_tape *un)
17607 {
17608         int rval = 0;
17609         int limit = st_retry_count;

17611         ST_FUNC(ST_DEVINFO, st_test_path_to_device);

17613         /*
17614          * XXX Newer drives may not RESEVATION CONFLICT a TUR.
17615          */
17616         do {
17617                 if (rval != 0) {
17618                         mutex_exit(ST_MUTEX);
17619                         delay(drv_sectohz(1));
17619                         delay(drv_usectohz(1000000));
17620                         mutex_enter(ST_MUTEX);
17621                 }
17622                 rval = st_rcmd(un, SCMD_TEST_UNIT_READY, 0, SYNC_CMD);
17623                 ST_RECOV(ST_DEVINFO, st_label, CE_NOTE,
17624                     "ping TUR returned 0x%x", rval);
17625                 limit--;
17626         } while (((rval == EACCES) || (rval == EBUSY)) && limit);

17628         if (un->un_status == KEY_NOT_READY || un->un_mediastate == MTIO_EJECTED)
17629                 rval = 0;

17631         return (rval);
17632 }
______unchanged_portion_omitted_
```

```
**********************************************************
   59626 Wed Aug 19 07:25:16 2015
new/usr/src/uts/common/io/sfe/sfe.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 *  sfe.c : DP83815/DP83816/SiS900 Fast Ethernet MAC driver for Solaris
 *
 * Copyright (c) 2002-2008 Masayuki Murayama.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the author nor the names of its contributors may be
 *    used to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */

/* Avoid undefined symbol for non IA architectures */
#pragma weak    inb
#pragma weak    outb

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * System Header files.
 */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/modctl.h>
#include <sys/errno.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/byteorder.h>
#include <sys/ethernet.h>
#include <sys/pci.h>

#include "sfe_mii.h"
#include "sfe_util.h"
#include "sfereg.h"
```



```
char    ident[] = "sis900/dp83815 driver v" "2.6.1t30os";

/* Debugging support */
#ifdef DEBUG_LEVEL
static int sfe_debug = DEBUG_LEVEL;
#if DEBUG_LEVEL > 4
#define CONS    "^"
#else
#define CONS    "!"
#endif
#define DPRINTF(n, args)        if (sfe_debug > (n)) cmn_err args
#else
#define CONS    "!"
#define DPRINTF(n, args)
#endif

/*
 * Useful macros and typedefs
 */
#define ONESEC          drv_sectohz(1)
#define ONESEC          (drv_usectohz(1*1000000))
#define ROUNDUP2(x, a)  (((x) + (a) - 1) & ~((a) - 1))

/*
 * Our configuration
 */
#define MAXTXFRAGS      1
#define MAXRXFRAGS      1

#ifndef TX_BUF_SIZE
#define TX_BUF_SIZE     64
#endif
#ifndef TX_RING_SIZE
#if MAXTXFRAGS == 1
#define TX_RING_SIZE    TX_BUF_SIZE
#else
#define TX_RING_SIZE    (TX_BUF_SIZE * 4)
#endif
#endif

#ifndef RX_BUF_SIZE
#define RX_BUF_SIZE     256
#endif
#ifndef RX_RING_SIZE
#define RX_RING_SIZE    RX_BUF_SIZE
#endif

#define OUR_INTR_BITS   \
        (ISR_DPERR | ISR_SSERR | ISR_RMABT | ISR_RTABT | ISR_RXSOVR |   \
        ISR_TXURN | ISR_TXDESC | ISR_TXERR |    \
        ISR_RXORN | ISR_RXIDLE | ISR_RXOK | ISR_RXERR)

#define USE_MULTICAST_HASHTBL

static int      sfe_tx_copy_thresh = 256;
static int      sfe_rx_copy_thresh = 256;

/* special PHY registers for SIS900 */
#define MII_CONFIG1     0x0010
#define MII_CONFIG2     0x0011
#define MII_MASK        0x0013
#define MII_RESV        0x0014

#define PHY_MASK                0xfffffff0
#define PHY_SIS900_INTERNAL     0x001d8000
#define PHY_ICS1893             0x0015f440
```

```
129 #define SFE_DESC_SIZE   16      /* including pads rounding up to power of 2 */

131 /*
132  * Supported chips
133  */
134 struct chip_info {
135         uint16_t        venid;
136         uint16_t        devid;
137         char            *chip_name;
138         int             chip_type;
139 #define CHIPTYPE_DP83815        0
140 #define CHIPTYPE_SIS900         1
141 };
```

unchanged_portion_omitted_

```
**********************************************************
  128018 Wed Aug 19 07:25:16 2015
new/usr/src/uts/common/io/sfe/sfe_util.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


4725 /*
4726  * Gem kstat support
4727  */

4729 #define GEM_LOCAL_DATA_SIZE(gc) \
4730         (sizeof (struct gem_dev) + \
4731         sizeof (struct mcast_addr) * GEM_MAXMC + \
4732         sizeof (struct txbuf) * ((gc)->gc_tx_buf_size) + \
4733         sizeof (void *) * ((gc)->gc_tx_buf_size))

4735 struct gem_dev *
4736 gem_do_attach(dev_info_t *dip, int port,
4737         struct gem_conf *gc, void *base, ddi_acc_handle_t *regs_handlep,
4738         void *lp, int lmsize)
4739 {
4740         struct gem_dev          *dp;
4741         int                     i;
4742         ddi_iblock_cookie_t     c;
4743         mac_register_t          *macp = NULL;
4744         int                     ret;
4745         int                     unit;
4746         int                     nports;

4748         unit = ddi_get_instance(dip);
4749         if ((nports = gc->gc_nports) == 0) {
4750                 nports = 1;
4751         }
4752         if (nports == 1) {
4753                 ddi_set_driver_private(dip, NULL);
4754         }

4756         DPRINTF(2, (CE_CONT, "!gem%d: gem_do_attach: called cmd:ATTACH",
4757             unit));

4759         /*
4760          * Allocate soft data structure
4761          */
4762         dp = kmem_zalloc(GEM_LOCAL_DATA_SIZE(gc), KM_SLEEP);

4764         if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
4765                 cmn_err(CE_WARN, "!gem%d: %s: mac_alloc failed",
4766                     unit, __func__);
4767                 return (NULL);
4768         }
4769         /* ddi_set_driver_private(dip, dp); */

4771         /* link to private area */
4772         dp->private = lp;
4773         dp->priv_size = lmsize;
4774         dp->mc_list = (struct mcast_addr *)&dp[1];

4776         dp->dip = dip;
4777         (void) sprintf(dp->name, gc->gc_name, nports * unit + port);

4779         /*
4780          * Get iblock cookie
4781          */
4782         if (ddi_get_iblock_cookie(dip, 0, &c) != DDI_SUCCESS) {
```



```
4783                 cmn_err(CE_CONT,
4784                     "!%s: gem_do_attach: ddi_get_iblock_cookie: failed",
4785                     dp->name);
4786                 goto err_free_private;
4787         }
4788         dp->iblock_cookie = c;

4790         /*
4791          * Initialize mutex's for this device.
4792          */
4793         mutex_init(&dp->intrlock, NULL, MUTEX_DRIVER, (void *)c);
4794         mutex_init(&dp->xmitlock, NULL, MUTEX_DRIVER, (void *)c);
4795         cv_init(&dp->tx_drain_cv, NULL, CV_DRIVER, NULL);

4797         /*
4798          * configure gem parameter
4799          */
4800         dp->base_addr = base;
4801         dp->regs_handle = *regs_handlep;
4802         dp->gc = *gc;
4803         gc = &dp->gc;
4804         /* patch for simplify dma resource management */
4805         gc->gc_tx_max_frags = 1;
4806         gc->gc_tx_max_descs_per_pkt = 1;
4807         gc->gc_tx_ring_size = gc->gc_tx_buf_size;
4808         gc->gc_tx_ring_limit = gc->gc_tx_buf_limit;
4809         gc->gc_tx_desc_write_oo = B_TRUE;

4811         gc->gc_nports = nports; /* fix nports */

4813         /* fix copy threadsholds */
4814         gc->gc_tx_copy_thresh = max(ETHERMIN, gc->gc_tx_copy_thresh);
4815         gc->gc_rx_copy_thresh = max(ETHERMIN, gc->gc_rx_copy_thresh);

4817         /* fix rx buffer boundary for iocache line size */
4818         ASSERT(gc->gc_dma_attr_txbuf.dma_attr_align-1 == gc->gc_tx_buf_align);
4819         ASSERT(gc->gc_dma_attr_rxbuf.dma_attr_align-1 == gc->gc_rx_buf_align);
4820         gc->gc_rx_buf_align = max(gc->gc_rx_buf_align, IOC_LINESIZE - 1);
4821         gc->gc_dma_attr_rxbuf.dma_attr_align = gc->gc_rx_buf_align + 1;

4823         /* fix descriptor boundary for cache line size */
4824         gc->gc_dma_attr_desc.dma_attr_align =
4825             max(gc->gc_dma_attr_desc.dma_attr_align, IOC_LINESIZE);

4827         /* patch get_packet method */
4828         if (gc->gc_get_packet == NULL) {
4829                 gc->gc_get_packet = &gem_get_packet_default;
4830         }

4832         /* patch get_rx_start method */
4833         if (gc->gc_rx_start == NULL) {
4834                 gc->gc_rx_start = &gem_rx_start_default;
4835         }

4837         /* calculate descriptor area */
4838         if (gc->gc_rx_desc_unit_shift >= 0) {
4839                 dp->rx_desc_size =
4840                     ROUNDUP(gc->gc_rx_ring_size << gc->gc_rx_desc_unit_shift,
4841                     gc->gc_dma_attr_desc.dma_attr_align);
4842         }
4843         if (gc->gc_tx_desc_unit_shift >= 0) {
4844                 dp->tx_desc_size =
4845                     ROUNDUP(gc->gc_tx_ring_size << gc->gc_tx_desc_unit_shift,
4846                     gc->gc_dma_attr_desc.dma_attr_align);
4847         }
```

```
4849         dp->mtu = ETHERMTU;
4850         dp->tx_buf = (void *)&dp->mc_list[GEM_MAXMC];
4851         /* link tx buffers */
4852         for (i = 0; i < dp->gc.gc_tx_buf_size; i++) {
4853                 dp->tx_buf[i].txb_next =
4854                     &dp->tx_buf[SLOT(i + 1, dp->gc.gc_tx_buf_size)];
4855         }

4857         dp->rxmode         = 0;
4858         dp->speed          = GEM_SPD_10;        /* default is 10Mbps */
4859         dp->full_duplex    = B_FALSE;           /* default is half */
4860         dp->flow_control   = FLOW_CONTROL_NONE;
4861         dp->poll_pkt_delay = 8;         /* typical coalease for rx packets */

4863         /* performance tuning parameters */
4864         dp->txthr    = ETHERMAX;        /* tx fifo threshold */
4865         dp->txmaxdma = 16*4;            /* tx max dma burst size */
4866         dp->rxthr    = 128;             /* rx fifo threshold */
4867         dp->rxmaxdma = 16*4;            /* rx max dma burst size */

4869         /*
4870          * Get media mode information from .conf file
4871          */
4872         gem_read_conf(dp);

4874         /* rx_buf_len is required buffer length without padding for alignment */
4875         dp->rx_buf_len = MAXPKTBUF(dp) + dp->gc.gc_rx_header_len;

4877         /*
4878          * Reset the chip
4879          */
4880         mutex_enter(&dp->intrlock);
4881         dp->nic_state = NIC_STATE_STOPPED;
4882         ret = (*dp->gc.gc_reset_chip)(dp);
4883         mutex_exit(&dp->intrlock);
4884         if (ret != GEM_SUCCESS) {
4885                 goto err_free_regs;
4886         }

4888         /*
4889          * HW dependant paremeter initialization
4890          */
4891         mutex_enter(&dp->intrlock);
4892         ret = (*dp->gc.gc_attach_chip)(dp);
4893         mutex_exit(&dp->intrlock);
4894         if (ret != GEM_SUCCESS) {
4895                 goto err_free_regs;
4896         }

4898 #ifdef DEBUG_MULTIFRAGS
4899         dp->gc.gc_tx_copy_thresh = dp->mtu;
4900 #endif
4901         /* allocate tx and rx resources */
4902         if (gem_alloc_memory(dp)) {
4903                 goto err_free_regs;
4904         }

4906         DPRINTF(0, (CE_CONT,
4907             "!%s: at 0x%x, %02x:%02x:%02x:%02x:%02x:%02x",
4908             dp->name, (long)dp->base_addr,
4909             dp->dev_addr.ether_addr_octet[0],
4910             dp->dev_addr.ether_addr_octet[1],
4911             dp->dev_addr.ether_addr_octet[2],
4912             dp->dev_addr.ether_addr_octet[3],
4913             dp->dev_addr.ether_addr_octet[4],
4914             dp->dev_addr.ether_addr_octet[5]));
```



```
4916         /* copy mac address */
4917         dp->cur_addr = dp->dev_addr;

4919         gem_gld3_init(dp, macp);

4921         /* Probe MII phy (scan phy) */
4922         dp->mii_lpable = 0;
4923         dp->mii_advert = 0;
4924         dp->mii_exp = 0;
4925         dp->mii_ctl1000 = 0;
4926         dp->mii_stat1000 = 0;
4927         if ((*dp->gc.gc_mii_probe)(dp) != GEM_SUCCESS) {
4928                 goto err_free_ring;
4929         }

4931         /* mask unsupported abilities */
4932         dp->anadv_autoneg &= BOOLEAN(dp->mii_status & MII_STATUS_CANAUTONEG);
4933         dp->anadv_1000fdx &=
4934             BOOLEAN(dp->mii_xstatus &
4935             (MII_XSTATUS_1000BASEX_FD | MII_XSTATUS_1000BASET_FD));
4936         dp->anadv_1000hdx &=
4937             BOOLEAN(dp->mii_xstatus &
4938             (MII_XSTATUS_1000BASEX | MII_XSTATUS_1000BASET));
4939         dp->anadv_100t4  &= BOOLEAN(dp->mii_status & MII_STATUS_100_BASE_T4);
4940         dp->anadv_100fdx &= BOOLEAN(dp->mii_status & MII_STATUS_100_BASEX_FD);
4941         dp->anadv_100hdx &= BOOLEAN(dp->mii_status & MII_STATUS_100_BASEX);
4942         dp->anadv_10fdx  &= BOOLEAN(dp->mii_status & MII_STATUS_10_FD);
4943         dp->anadv_10hdx  &= BOOLEAN(dp->mii_status & MII_STATUS_10);

4945         gem_choose_forcedmode(dp);

4947         /* initialize MII phy if required */
4948         if (dp->gc.gc_mii_init) {
4949                 if ((*dp->gc.gc_mii_init)(dp) != GEM_SUCCESS) {
4950                         goto err_free_ring;
4951                 }
4952         }

4954         /*
4955          * initialize kstats including mii statistics
4956          */
4957         gem_nd_setup(dp);

4959         /*
4960          * Add interrupt to system.
4961          */
4962         if (ret = mac_register(macp, &dp->mh)) {
4963                 cmn_err(CE_WARN, "!%s: mac_register failed, error:%d",
4964                     dp->name, ret);
4965                 goto err_release_stats;
4966         }
4967         mac_free(macp);
4968         macp = NULL;

4970         if (dp->misc_flag & GEM_SOFTINTR) {
4971                 if (ddi_add_softintr(dip,
4972                     DDI_SOFTINT_LOW, &dp->soft_id,
4973                     NULL, NULL,
4974                     (uint_t (*)(caddr_t))gem_intr,
4975                     (caddr_t)dp) != DDI_SUCCESS) {
4976                         cmn_err(CE_WARN, "!%s: ddi_add_softintr failed",
4977                             dp->name);
4978                         goto err_unregister;
4979                 }
4980         } else if ((dp->misc_flag & GEM_NOINTR) == 0) {
```

```
4981                 if (ddi_add_intr(dip, 0, NULL, NULL,
4982                     (uint_t (*)(caddr_t))gem_intr,
4983                     (caddr_t)dp) != DDI_SUCCESS) {
4984                         cmn_err(CE_WARN, "!%s: ddi_add_intr failed", dp->name);
4985                         goto err_unregister;
4986                 }
4987         } else {
4988                 /*
4989                  * Dont use interrupt.
4990                  * schedule first call of gem_intr_watcher
4991                  */
4992                 dp->intr_watcher_id =
4993                     timeout((void (*)(void *))gem_intr_watcher,
4994                     (void *)dp, drv_sectohz(3));
4994                     (void *)dp, drv_usectohz(3*1000000));
4995         }

4997         /* link this device to dev_info */
4998         dp->next = (struct gem_dev *)ddi_get_driver_private(dip);
4999         dp->port = port;
5000         ddi_set_driver_private(dip, (caddr_t)dp);

5002         /* reset mii phy and start mii link watcher */
5003         gem_mii_start(dp);

5005         DPRINTF(2, (CE_CONT, "!gem_do_attach: return: success"));
5006         return (dp);

5008 err_unregister:
5009         (void) mac_unregister(dp->mh);
5010 err_release_stats:
5011         /* release NDD resources */
5012         gem_nd_cleanup(dp);

5014 err_free_ring:
5015         gem_free_memory(dp);
5016 err_free_regs:
5017         ddi_regs_map_free(&dp->regs_handle);
5018 err_free_locks:
5019         mutex_destroy(&dp->xmitlock);
5020         mutex_destroy(&dp->intrlock);
5021         cv_destroy(&dp->tx_drain_cv);
5022 err_free_private:
5023         if (macp) {
5024                 mac_free(macp);
5025         }
5026         kmem_free((caddr_t)dp, GEM_LOCAL_DATA_SIZE(gc));

5028         return (NULL);
5029 }
______unchanged_portion_omitted_

5159 int
5160 gem_resume(dev_info_t *dip)
5161 {
5162         struct gem_dev  *dp;

5164         /*
5165          * restart the device
5166          */
5167         dp = GEM_GET_DEV(dip);
5168         ASSERT(dp);

5170         DPRINTF(0, (CE_CONT, "!%s: %s: called", dp->name, __func__));

5172         for (; dp; dp = dp->next) {
```



```
5174                 /*
5175                  * Bring up the nic after power up
5176                  */

5178                 /* gem_xxx.c layer to setup power management state. */
5179                 ASSERT(!dp->mac_active);

5181                 /* reset the chip, because we are just after power up. */
5182                 mutex_enter(&dp->intrlock);

5184                 dp->mac_suspended = B_FALSE;
5185                 dp->nic_state = NIC_STATE_STOPPED;

5187                 if ((*dp->gc.gc_reset_chip)(dp) != GEM_SUCCESS) {
5188                         cmn_err(CE_WARN, "%s: %s: failed to reset chip",
5189                             dp->name, __func__);
5190                         mutex_exit(&dp->intrlock);
5191                         goto err;
5192                 }
5193                 mutex_exit(&dp->intrlock);

5195                 /* initialize mii phy because we are just after power up */
5196                 if (dp->gc.gc_mii_init) {
5197                         (void) (*dp->gc.gc_mii_init)(dp);
5198                 }

5200                 if (dp->misc_flag & GEM_NOINTR) {
5201                         /*
5202                          * schedule first call of gem_intr_watcher
5203                          * instead of interrupts.
5204                          */
5205                         dp->intr_watcher_id =
5206                             timeout((void (*)(void *))gem_intr_watcher,
5207                             (void *)dp, drv_sectohz(3));
5207                             (void *)dp, drv_usectohz(3*1000000));
5208                 }

5210                 /* restart mii link watcher */
5211                 gem_mii_start(dp);

5213                 /* restart mac */
5214                 mutex_enter(&dp->intrlock);

5216                 if (gem_mac_init(dp) != GEM_SUCCESS) {
5217                         mutex_exit(&dp->intrlock);
5218                         goto err_reset;
5219                 }
5220                 dp->nic_state = NIC_STATE_INITIALIZED;

5222                 /* setup media mode if the link have been up */
5223                 if (dp->mii_state == MII_STATE_LINKUP) {
5224                         if ((dp->gc.gc_set_media)(dp) != GEM_SUCCESS) {
5225                                 mutex_exit(&dp->intrlock);
5226                                 goto err_reset;
5227                         }
5228                 }

5230                 /* enable mac address and rx filter */
5231                 dp->rxmode |= RXMODE_ENABLE;
5232                 if ((*dp->gc.gc_set_rx_filter)(dp) != GEM_SUCCESS) {
5233                         mutex_exit(&dp->intrlock);
5234                         goto err_reset;
5235                 }
5236                 dp->nic_state = NIC_STATE_ONLINE;
```

```
5238                 /* restart tx timeout watcher */
5239                 dp->timeout_id = timeout((void (*)(void *))gem_tx_timeout,
5240                     (void *)dp,
5241                     dp->gc.gc_tx_timeout_interval);

5243                 /* now the nic is fully functional */
5244                 if (dp->mii_state == MII_STATE_LINKUP) {
5245                         if (gem_mac_start(dp) != GEM_SUCCESS) {
5246                                 mutex_exit(&dp->intrlock);
5247                                 goto err_reset;
5248                         }
5249                 }
5250                 mutex_exit(&dp->intrlock);
5251         }

5253         return (DDI_SUCCESS);

5255 err_reset:
5256         if (dp->intr_watcher_id) {
5257                 while (untimeout(dp->intr_watcher_id) == -1)
5258                         ;
5259                 dp->intr_watcher_id = 0;
5260         }
5261         mutex_enter(&dp->intrlock);
5262         (*dp->gc.gc_reset_chip)(dp);
5263         dp->nic_state = NIC_STATE_STOPPED;
5264         mutex_exit(&dp->intrlock);

5266 err:
5267         return (DDI_FAILURE);
5268 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17855 Wed Aug 19 07:25:17 2015
new/usr/src/uts/common/io/sfe/sfe_util.h
XXXX introduce drv_sectohz
**********************************************************
```

```
   1 /*
   2  *  sfe_util.h: header to support the gem layer used by Masa Murayama
   3  *
   4  * Copyright (c) 2002-2008 Masayuki Murayama.  All rights reserved.
   5  *
   6  * Redistribution and use in source and binary forms, with or without
   7  * modification, are permitted provided that the following conditions are met:
   8  *
   9  * 1. Redistributions of source code must retain the above copyright notice,
  10  *    this list of conditions and the following disclaimer.
  11  *
  12  * 2. Redistributions in binary form must reproduce the above copyright notice,
  13  *    this list of conditions and the following disclaimer in the documentation
  14  *    and/or other materials provided with the distribution.
  15  *
  16  * 3. Neither the name of the author nor the names of its contributors may be
  17  *    used to endorse or promote products derived from this software without
  18  *    specific prior written permission.
  19  *
  20  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  21  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  22  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  23  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
  24  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
  25  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
  26  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
  27  * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
  28  * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  29  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
  30  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
  31  * DAMAGE.
  32  */

  34 /*
  35  * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  36  * Use is subject to license terms.
  37  */

  39 #ifndef _SFE_UTIL_H_
  40 #define _SFE_UTIL_H_
  41 #include <sys/mac_provider.h>
  42 #include <sys/mac_ether.h>

  44 /*
  45  * Useful macros and typedefs
  46  */

  48 #define GEM_NAME_LEN    32

  50 #define GEM_TX_TIMEOUT          drv_sectohz(5)
  51 #define GEM_TX_TIMEOUT_INTERVAL drv_sectohz(1)
  52 #define GEM_LINK_WATCH_INTERVAL drv_sectohz(1)
  50 #define GEM_TX_TIMEOUT          (drv_usectohz(5*1000000))
  51 #define GEM_TX_TIMEOUT_INTERVAL (drv_usectohz(1*1000000))
  52 #define GEM_LINK_WATCH_INTERVAL (drv_usectohz(1*1000000))       /* 1 sec */

  54 /* general return code */
  55 #define GEM_SUCCESS     0
  56 #define GEM_FAILURE     (-1)

  58 /* return code of gem_tx_done */
```



```
  59 #define INTR_RESTART_TX 0x80000000

  61 typedef int32_t         seqnum_t;

  63 /*
  64  * I/O instructions
  65  */
  66 #define OUTB(dp, p, v)  \
  67         ddi_put8((dp)->regs_handle, \
  68                 (void *)((caddr_t)((dp)->base_addr) + (p)), v)
  69 #define OUTW(dp, p, v)  \
  70         ddi_put16((dp)->regs_handle, \
  71                 (void *)((caddr_t)((dp)->base_addr) + (p)), v)
  72 #define OUTL(dp, p, v)  \
  73         ddi_put32((dp)->regs_handle, \
  74             (void *)((caddr_t)((dp)->base_addr) + (p)), v)
  75 #define OUTLINL(dp, p, v) \
  76         ddi_put32((dp)->regs_handle, \
  77             (void *)((caddr_t)((dp)->base_addr) + (p)), v); \
  78         (void) INL((dp), (p))
  79 #define INB(dp, p)      \
  80         ddi_get8((dp)->regs_handle, \
  81                 (void *)(((caddr_t)(dp)->base_addr) + (p)))
  82 #define INW(dp, p)      \
  83         ddi_get16((dp)->regs_handle, \
  84                 (void *)(((caddr_t)(dp)->base_addr) + (p)))
  85 #define INL(dp, p)      \
  86         ddi_get32((dp)->regs_handle, \
  87                 (void *)(((caddr_t)(dp)->base_addr) + (p)))

  89 struct gem_stats {
  90         uint32_t        intr;

  92         uint32_t        crc;
  93         uint32_t        errrcv;
  94         uint32_t        overflow;
  95         uint32_t        frame;
  96         uint32_t        missed;
  97         uint32_t        runt;
  98         uint32_t        frame_too_long;
  99         uint32_t        norcvbuf;
 100         uint32_t        sqe;

 102         uint32_t        collisions;
 103         uint32_t        first_coll;
 104         uint32_t        multi_coll;
 105         uint32_t        excoll;
 106         uint32_t        xmit_internal_err;
 107         uint32_t        nocarrier;
 108         uint32_t        defer;
 109         uint32_t        errxmt;
 110         uint32_t        underflow;
 111         uint32_t        xmtlatecoll;
 112         uint32_t        noxmtbuf;
 113         uint32_t        jabber;

 115         uint64_t        rbytes;
 116         uint64_t        obytes;
 117         uint64_t        rpackets;
 118         uint64_t        opackets;
 119         uint32_t        rbcast;
 120         uint32_t        obcast;
 121         uint32_t        rmcast;
 122         uint32_t        omcast;
 123         uint32_t        rcv_internal_err;
 124 };
______unchanged_portion_omitted_
```

```
**********************************************************
  124094 Wed Aug 19 07:25:17 2015
new/usr/src/uts/common/io/skd/skd.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 192 /*
 193  * Solaris module loading/unloading routines
 194  */

 196 /*
 197  *
 198  * Name:        _init, performs initial installation
 199  *
 200  * Inputs:      None.
 201  *
 202  * Returns:     Returns the value returned by the ddi_softstate_init function
 203  *              on a failure to create the device state structure or the result
 204  *              of the module install routines.
 205  *
 206  */
 207 int
 208 _init(void)
 209 {
 210         int             rval = 0;
 211         int             tgts = 0;

 213         tgts |= 0x02;
 214         tgts |= 0x08;   /* In #ifdef NEXENTA block from original sTec drop. */

 216         /*
 217          * drv_usectohz() is a function, so can't initialize it at
 218          * instantiation.
 219          */
 220         skd_timer_ticks = drv_sectohz(1);
 220         skd_timer_ticks = drv_usectohz(1000000);

 222         Dcmn_err(CE_NOTE,
 223             "<# Installing skd Driver dbg-lvl=%d %s %x>",
 224             skd_dbg_level, DRV_BUILD_ID, tgts);

 226         rval = ddi_soft_state_init(&skd_state, sizeof (skd_device_t), 0);
 227         if (rval != DDI_SUCCESS)
 228                 return (rval);

 230         bd_mod_init(&skd_dev_ops);

 232         rval = mod_install(&modlinkage);
 233         if (rval != DDI_SUCCESS) {
 234                 ddi_soft_state_fini(&skd_state);
 235                 bd_mod_fini(&skd_dev_ops);
 236         }

 238         return (rval);
 239 }
______unchanged_portion_omitted_

2859 /*
2860  *
2861  * Name:        skd_stop_device, stops the device.
2862  *
2863  * Inputs:      skdev           - device state structure.
2864  *
2865  * Returns:     Nothing.
2866  *
```



```
2867  */
2868 static void
2869 skd_stop_device(struct skd_device *skdev)
2870 {
2871         clock_t cur_ticks, tmo;
2872         int secs;
2873         struct skd_special_context *skspcl = &skdev->internal_skspcl;

2875         if (SKD_DRVR_STATE_ONLINE != skdev->state) {
2876                 Dcmn_err(CE_NOTE, "(%s): skd_stop_device not online no sync\n",
2877                     skdev->name);
2878                 goto stop_out;
2879         }

2881         if (SKD_REQ_STATE_IDLE != skspcl->req.state) {
2882                 Dcmn_err(CE_NOTE, "(%s): skd_stop_device no special\n",
2883                     skdev->name);
2884                 goto stop_out;
2885         }

2887         skdev->state = SKD_DRVR_STATE_SYNCING;
2888         skdev->sync_done = 0;

2890         skd_send_internal_skspcl(skdev, skspcl, SYNCHRONIZE_CACHE);

2892         secs = 10;
2893         mutex_enter(&skdev->skd_internalio_mutex);
2894         while (skdev->sync_done == 0) {
2895                 cur_ticks = ddi_get_lbolt();
2896                 tmo = cur_ticks + drv_sectohz(secs);
2896                 tmo = cur_ticks + drv_usectohz(1000000 * secs);
2897                 if (cv_timedwait(&skdev->cv_waitq,
2898                     &skdev->skd_internalio_mutex, tmo) == -1) {
2899                         /* Oops - timed out */

2901                         Dcmn_err(CE_NOTE, "stop_device - %d secs TMO", secs);
2902                 }
2903         }

2905         mutex_exit(&skdev->skd_internalio_mutex);

2907         switch (skdev->sync_done) {
2908         case 0:
2909                 Dcmn_err(CE_NOTE, "(%s): skd_stop_device no sync\n",
2910                     skdev->name);
2911                 break;
2912         case 1:
2913                 Dcmn_err(CE_NOTE, "(%s): skd_stop_device sync done\n",
2914                     skdev->name);
2915                 break;
2916         default:
2917                 Dcmn_err(CE_NOTE, "(%s): skd_stop_device sync error\n",
2918                     skdev->name);
2919         }


2922 stop_out:
2923         skdev->state = SKD_DRVR_STATE_STOPPING;

2925         skd_disable_interrupts(skdev);

2927         /* ensure all ints on device are cleared */
2928         SKD_WRITEL(skdev, FIT_INT_DEF_MASK, FIT_INT_STATUS_HOST);
2929         /* soft reset the device to unload with a clean slate */
2930         SKD_WRITEL(skdev, FIT_CR_SOFT_RESET, FIT_CONTROL);
2931 }
______unchanged_portion_omitted_
```

```
4502 /*
4503  *
4504  * Name:        skd_wait_for_s1120, wait for device to finish
4505  *              its initialization.
4506  *
4507  * Inputs:      skdev           - device state structure.
4508  *
4509  * Returns:     DDI_SUCCESS or DDI_FAILURE.
4510  *
4511  */
4512 static int
4513 skd_wait_for_s1120(skd_device_t *skdev)
4514 {
4515         clock_t cur_ticks, tmo;
4516         int     loop_cntr = 0;
4517         int     rc = DDI_FAILURE;

4519         mutex_enter(&skdev->skd_internalio_mutex);

4521         while (skdev->gendisk_on == 0) {
4522                 cur_ticks = ddi_get_lbolt();
4523                 tmo = cur_ticks + drv_sectohz(1);
4523                 tmo = cur_ticks + drv_usectohz(MICROSEC);
4524                 if (cv_timedwait(&skdev->cv_waitq,
4525                     &skdev->skd_internalio_mutex, tmo) == -1) {
4526                         /* Oops - timed out */
4527                         if (loop_cntr++ > 10)
4528                                 break;
4529                 }
4530         }

4532         mutex_exit(&skdev->skd_internalio_mutex);

4534         if (skdev->gendisk_on == 1)
4535                 rc = DDI_SUCCESS;

4537         return (rc);
4538 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15156 Wed Aug 19 07:25:17 2015
new/usr/src/uts/common/io/srn.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_
 510 /*
 511  * A very simple handshake with the srn driver,
 512  * only one outstanding event at a time.
 513  * The OS delivers the event and depending on type,
 514  * either blocks waiting for the ack, or drives on
 515  */
 516 void
 517 srn_notify(int type, int event)
 518 {
 519         int clone, count;
 520         PMD(PMD_SX, ("srn_notify entered with type %d, event 0x%x\n",
 521             type, event));
 522         ASSERT(mutex_owned(&srn_clone_lock));
 523         switch (type) {
 524         case SRN_TYPE_APM:
 525                 if (srn_apm_count == 0) {
 526                         PMD(PMD_SX, ("no apm types\n"))
 527                         return;
 528                 }
 529                 count = srn_apm_count;
 530                 break;
 531         case SRN_TYPE_AUTOSX:
 532                 if (srn_autosx_count == 0) {
 533                         PMD(PMD_SX, ("no autosx types\n"))
 534                         return;
 535                 }
 536                 count = srn_autosx_count;
 537                 break;
 538         default:
 539                 ASSERT(0);
 540                 break;
 541         }
 542         ASSERT(count > 0);
 543         PMD(PMD_SX, ("count %d\n", count))
 544         for (clone = 0; clone < SRN_MAX_CLONE; clone++) {
 545                 if (srn.srn_type[clone] == type) {
 546 #ifdef DEBUG
 547                         if (type == SRN_TYPE_APM && !srn.srn_fault[clone]) {
 548                                 ASSERT(srn.srn_pending[clone].ae_type == 0);
 549                                 ASSERT(srn_poll_cnt[clone] == 0);
 550                                 ASSERT(srn.srn_delivered[clone] == 0);
 551                         }
 552 #endif
 553                         srn.srn_pending[clone].ae_type = event;
 554                         srn_poll_cnt[clone] = 1;
 555                         PMD(PMD_SX, ("pollwake %d\n", clone))
 556                         pollwakeup(&srn_pollhead[clone], (POLLRDNORM | POLLIN));
 557                         count--;
 558                         if (count == 0)
 559                                 break;
 560                 }
 561         }
 562         if (type == SRN_TYPE_AUTOSX) {          /* we don't wait */
 563                 PMD(PMD_SX, ("Not waiting for AUTOSX ack\n"))
 564                 return;
 565         }
 566         ASSERT(type == SRN_TYPE_APM);
 567         /* otherwise wait for acks */
 568 restart:
 569         /*
```



```
 570          * We wait until all of the pending events are cleared.
 571          * We have to start over every time we do a cv_wait because
 572          * we give up the mutex and can be re-entered
 573          */
 574         for (clone = 1; clone < SRN_MAX_CLONE; clone++) {
 575                 if (srn.srn_clones[clone] == 0 ||
 576                     srn.srn_type[clone] != SRN_TYPE_APM)
 577                         continue;
 578                 if (srn.srn_pending[clone].ae_type && !srn.srn_fault[clone]) {
 579                         PMD(PMD_SX, ("srn_notify waiting for ack for clone %d, "
 580                             "event %x\n", clone, event))
 581                         if (cv_timedwait(&srn_clones_cv[clone],
 582                             &srn_clone_lock, ddi_get_lbolt() +
 583                             drv_sectohz(srn_timeout)) == -1) {
 583                             drv_usectohz(srn_timeout * 1000000)) == -1) {
 584                                 /*
 585                                  * Client didn't respond, mark it as faulted
 586                                  * and continue as if a regular signal.
 587                                  */
 588                                 PMD(PMD_SX, ("srn_notify: clone %d did not "
 589                                     "ack event %x\n", clone, event))
 590                                 cmn_err(CE_WARN, "srn_notify: clone %d did "
 591                                     "not ack event %x\n", clone, event);
 592                                 srn.srn_fault[clone] = 1;
 593                         }
 594                         goto restart;
 595                 }
 596         }
 597         PMD(PMD_SX, ("srn_notify done with %x\n", event))
 598 }
______unchanged_portion_omitted_
```

****
75240 Wed Aug 19 07:25:17 2015
new/usr/src/uts/common/io/usb/clients/usbser/usbser.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
1755 /*
1756  * during close, wait until pending data is gone or the signal is sent
1757  */
1758 static void
1759 usbser_close_drain(usbser_port_t *pp)
1760 {
1761         int     need_drain;
1762         clock_t until;
1763         int     rval = USB_SUCCESS;

1765         /*
1766          * port_wq_data_cnt indicates amount of data on the write queue,
1767          * which becomes zero when all data is submitted to DSD. But usbser
1768          * stays busy until it gets tx callback from DSD, signalling that
1769          * data has been sent over USB. To be continued in the next comment...
1770          */
1771         until = ddi_get_lbolt() +
1772             drv_sectohz(USBSER_WQ_DRAIN_TIMEOUT);
1772             drv_usectohz(USBSER_WQ_DRAIN_TIMEOUT * 1000000);

1774         while ((pp->port_wq_data_cnt > 0) && USBSER_PORT_IS_BUSY(pp)) {
1775                 if ((rval = cv_timedwait_sig(&pp->port_act_cv, &pp->port_mutex,
1776                     until)) <= 0) {

1778                         break;
1779                 }
1780         }

1782         /* don't drain if timed out or received a signal */
1783         need_drain = (pp->port_wq_data_cnt == 0) || !USBSER_PORT_IS_BUSY(pp) ||
1784             (rval != USB_SUCCESS);

1786         mutex_exit(&pp->port_mutex);
1787         /*
1788          * Once the data reaches USB serial box, it may still be stored in its
1789          * internal output buffer (FIFO). We call DSD drain to ensure that all
1790          * the data is transmitted transmitted over the serial line.
1791          */
1792         if (need_drain) {
1793                 rval = USBSER_DS_FIFO_DRAIN(pp, USBSER_TX_FIFO_DRAIN_TIMEOUT);
1794                 if (rval != USB_SUCCESS) {
1795                         (void) USBSER_DS_FIFO_FLUSH(pp, DS_TX);
1796                 }
1797         } else {
1798                 (void) USBSER_DS_FIFO_FLUSH(pp, DS_TX);
1799         }
1800         mutex_enter(&pp->port_mutex);
1801 }
```

_____unchanged_portion_omitted_

****
74679 Wed Aug 19 07:25:17 2015
new/usr/src/uts/common/io/usb/clients/usbser/usbser_keyspan/keyspan_dsd.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
2521 /*
2522  * wait until local tx buffer drains.
2523  * 'timeout' is in seconds, zero means wait forever
2524  */
2525 static int
2526 keyspan_wait_tx_drain(keyspan_port_t *kp, int timeout)
2527 {
2528         clock_t until;
2529         int     over = 0;

2531         USB_DPRINTF_L4(DPRINT_OUT_DATA, kp->kp_lh, "keyspan_wait_tx_drain:"
2532             "timeout = %d", timeout);
2533         until = ddi_get_lbolt() + drv_sectohz(timeout);
2533         until = ddi_get_lbolt() + drv_usectohz(1000000 * timeout);

2535         while (kp->kp_tx_mp && !over) {
2536                 if (timeout > 0) {
2537                         over = (cv_timedwait_sig(&kp->kp_tx_cv,
2538                             &kp->kp_mutex, until) <= 0);
2539                 } else {
2540                         over = (cv_wait_sig(&kp->kp_tx_cv, &kp->kp_mutex) == 0);
2541                 }
2542         }

2544         return ((kp->kp_tx_mp == NULL) ? USB_SUCCESS : USB_FAILURE);
2545 }
```

_____***unchanged_portion_omitted_***

****
37262 Wed Aug 19 07:25:18 2015
new/usr/src/uts/common/io/usb/hcd/ehci/ehci_isoch.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 258 /*
 259  * ehci_wait_for_transfers_completion:
 260  *
 261  * Wait for processing all completed transfers and to send results
 262  * to upstream.
 263  */
 264 static void
 265 ehci_wait_for_isoc_completion(
 266         ehci_state_t            *ehcip,
 267         ehci_pipe_private_t     *pp)
 268 {
 269         ASSERT(mutex_owned(&ehcip->ehci_int_mutex));

 271         if (pp->pp_itw_head == NULL) {

 273                 return;
 274         }

 276         (void) cv_reltimedwait(&pp->pp_xfer_cmpl_cv, &ehcip->ehci_int_mutex,
 277             drv_sectohz(EHCI_XFER_CMPL_TIMEWAIT), TR_CLOCK_TICK);
 277             drv_usectohz(EHCI_XFER_CMPL_TIMEWAIT * 1000000), TR_CLOCK_TICK);

 279         if (pp->pp_itw_head) {
 280                 USB_DPRINTF_L2(PRINT_MASK_LISTS, ehcip->ehci_log_hdl,
 281                     "ehci_wait_for_isoc_completion: "
 282                     "No transfers completion confirmation received");
 283         }
 284 }
```

_____***unchanged_portion_omitted_***

```
***********************************************************
  106017 Wed Aug 19 07:25:18 2015
new/usr/src/uts/common/io/usb/hcd/ehci/ehci_xfer.c
XXXX introduce drv_sectohz
***********************************************************
______unchanged_portion_omitted_


3374 /*
3375  * ehci_start_timer:
3376  *
3377  * Start the pipe's timer
3378  */
3379 static void
3380 ehci_start_timer(
3381         ehci_state_t            *ehcip,
3382         ehci_pipe_private_t     *pp)
3383 {
3384         USB_DPRINTF_L4(PRINT_MASK_LISTS,  ehcip->ehci_log_hdl,
3385             "ehci_start_timer: ehcip = 0x%p, pp = 0x%p",
3386             (void *)ehcip, (void *)pp);

3388         ASSERT(mutex_owned(&ehcip->ehci_int_mutex));

3390         /*
3391          * Start the pipe's timer only if currently timer is not
3392          * running and if there are any transfers on the timeout
3393          * list. This timer will be per pipe.
3394          */
3395         if ((!pp->pp_timer_id) && (pp->pp_timeout_list)) {
3396                 pp->pp_timer_id = timeout(ehci_xfer_timeout_handler,
3397                     (void *)(pp->pp_pipe_handle), drv_sectohz(1));
3397                     (void *)(pp->pp_pipe_handle), drv_usectohz(1000000));
3398         }
3399 }
______unchanged_portion_omitted_


3744 /*
3745  * ehci_wait_for_transfers_completion:
3746  *
3747  * Wait for processing all completed transfers and to send results
3748  * to upstream.
3749  */
3750 static void
3751 ehci_wait_for_transfers_completion(
3752         ehci_state_t            *ehcip,
3753         ehci_pipe_private_t     *pp)
3754 {
3755         ehci_trans_wrapper_t    *next_tw = pp->pp_tw_head;
3756         ehci_qtd_t              *qtd;

3758         USB_DPRINTF_L4(PRINT_MASK_LISTS,
3759             ehcip->ehci_log_hdl,
3760             "ehci_wait_for_transfers_completion: pp = 0x%p", (void *)pp);

3762         ASSERT(mutex_owned(&ehcip->ehci_int_mutex));

3764         if ((ehci_state_is_operational(ehcip)) != USB_SUCCESS) {

3766                 return;
3767         }

3769         pp->pp_count_done_qtds = 0;

3771         /* Process the transfer wrappers for this pipe */
```



```
3772         while (next_tw) {
3773                 qtd = (ehci_qtd_t *)next_tw->tw_qtd_head;

3775                 /*
3776                  * Walk through each QTD for this transfer wrapper.
3777                  * If a QTD still exists, then it is either on done
3778                  * list or on the QH's list.
3779                  */
3780                 while (qtd) {
3781                         if (!(Get_QTD(qtd->qtd_ctrl) &
3782                             EHCI_QTD_CTRL_ACTIVE_XACT)) {
3783                                 pp->pp_count_done_qtds++;
3784                         }

3786                         qtd = ehci_qtd_iommu_to_cpu(ehcip,
3787                             Get_QTD(qtd->qtd_tw_next_qtd));
3788                 }

3790                 next_tw = next_tw->tw_next;
3791         }

3793         USB_DPRINTF_L3(PRINT_MASK_LISTS, ehcip->ehci_log_hdl,
3794             "ehci_wait_for_transfers_completion: count_done_qtds = 0x%x",
3795             pp->pp_count_done_qtds);

3797         if (!pp->pp_count_done_qtds) {

3799                 return;
3800         }

3802         (void) cv_reltimedwait(&pp->pp_xfer_cmpl_cv, &ehcip->ehci_int_mutex,
3803             drv_sectohz(EHCI_XFER_CMPL_TIMEWAIT), TR_CLOCK_TICK);
3803             drv_usectohz(EHCI_XFER_CMPL_TIMEWAIT * 1000000), TR_CLOCK_TICK);

3805         if (pp->pp_count_done_qtds) {

3807                 USB_DPRINTF_L2(PRINT_MASK_LISTS, ehcip->ehci_log_hdl,
3808                     "ehci_wait_for_transfers_completion:"
3809                     "No transfers completion confirmation received");
3810         }
3811 }
______unchanged_portion_omitted_
```

```
**********************************************************
  296109 Wed Aug 19 07:25:18 2015
new/usr/src/uts/common/io/usb/hcd/openhci/ohci.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


1506 /*
1507  * ohci_init_ctlr:
1508  *
1509  * Initialize the Host Controller (HC).
1510  */
1511 static int
1512 ohci_init_ctlr(ohci_state_t     *ohcip)
1513 {
1514         int                     revision, curr_control, max_packet = 0;
1515         clock_t                 sof_time_wait;
1516         int                     retry = 0;
1517         int                     ohci_frame_interval;

1519         USB_DPRINTF_L4(PRINT_MASK_ATTA, ohcip->ohci_log_hdl, "ohci_init_ctlr:");

1521         if (ohci_take_control(ohcip) != DDI_SUCCESS) {
1522                 USB_DPRINTF_L2(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1523                     "ohci_init_ctlr: ohci_take_control failed\n");

1525                 return (DDI_FAILURE);
1526         }

1528         /*
1529          * Soft reset the host controller.
1530          *
1531          * On soft reset, the ohci host controller moves to the
1532          * USB Suspend state in which most of the ohci operational
1533          * registers are reset except stated ones. The soft reset
1534          * doesn't cause a reset to the ohci root hub and even no
1535          * subsequent reset signaling should be asserterd to its
1536          * down stream.
1537          */
1538         Set_OpReg(hcr_cmd_status, HCR_STATUS_RESET);

1540         mutex_exit(&ohcip->ohci_int_mutex);
1541         /* Wait 10ms for reset to complete */
1542         delay(drv_usectohz(OHCI_RESET_TIMEWAIT));
1543         mutex_enter(&ohcip->ohci_int_mutex);

1545         /*
1546          * Do hard reset the host controller.
1547          *
1548          * Now perform USB reset in order to reset the ohci root
1549          * hub.
1550          */
1551         Set_OpReg(hcr_control, HCR_CONTROL_RESET);

1553         /*
1554          * According to Section 5.1.2.3 of the specification, the
1555          * host controller will go into suspend state immediately
1556          * after the reset.
1557          */

1559         /* Verify the version number */
1560         revision = Get_OpReg(hcr_revision);

1562         if ((revision & HCR_REVISION_MASK) != HCR_REVISION_1_0) {
```



```
1564                 return (DDI_FAILURE);
1565         }

1567         USB_DPRINTF_L4(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1568             "ohci_init_ctlr: Revision verified");

1570         /* hcca area need not be initialized on resume */
1571         if (ohcip->ohci_hc_soft_state == OHCI_CTLR_INIT_STATE) {

1573                 /* Initialize the hcca area */
1574                 if (ohci_init_hcca(ohcip) != DDI_SUCCESS) {

1576                         return (DDI_FAILURE);
1577                 }
1578         }

1580         /*
1581          * Workaround for ULI1575 chipset. Following OHCI Operational Memory
1582          * Registers are not cleared to their default value on reset.
1583          * Explicitly set the registers to default value.
1584          */
1585         if (ohcip->ohci_vendor_id == PCI_ULI1575_VENID &&
1586             ohcip->ohci_device_id == PCI_ULI1575_DEVID) {
1587                 Set_OpReg(hcr_control, HCR_CONTROL_DEFAULT);
1588                 Set_OpReg(hcr_intr_enable, HCR_INT_ENABLE_DEFAULT);
1589                 Set_OpReg(hcr_HCCA, HCR_HCCA_DEFAULT);
1590                 Set_OpReg(hcr_ctrl_head, HCR_CONTROL_HEAD_ED_DEFAULT);
1591                 Set_OpReg(hcr_bulk_head, HCR_BULK_HEAD_ED_DEFAULT);
1592                 Set_OpReg(hcr_frame_interval, HCR_FRAME_INTERVAL_DEFAULT);
1593                 Set_OpReg(hcr_periodic_strt, HCR_PERIODIC_START_DEFAULT);
1594         }

1596         /* Set the HcHCCA to the physical address of the HCCA block */
1597         Set_OpReg(hcr_HCCA, (uint_t)ohcip->ohci_hcca_cookie.dmac_address);

1599         /*
1600          * Set HcInterruptEnable to enable all interrupts except Root
1601          * Hub Status change and SOF interrupts.
1602          */
1603         Set_OpReg(hcr_intr_enable, HCR_INTR_SO | HCR_INTR_WDH |
1604             HCR_INTR_RD | HCR_INTR_UE | HCR_INTR_FNO | HCR_INTR_MIE);

1606         /*
1607          * For non-periodic transfers, reserve atleast for one low-speed
1608          * device transaction. According to USB Bandwidth Analysis white
1609          * paper and also as per OHCI Specification 1.0a, section 7.3.5,
1610          * page 123, one low-speed transaction takes 0x628h full speed
1611          * bits (197 bytes), which comes to around 13% of USB frame time.
1612          *
1613          * The periodic transfers will get around 87% of USB frame time.
1614          */
1615         Set_OpReg(hcr_periodic_strt,
1616             ((PERIODIC_XFER_STARTS * BITS_PER_BYTE) - 1));

1618         /* Save the contents of the Frame Interval Registers */
1619         ohcip->ohci_frame_interval = Get_OpReg(hcr_frame_interval);

1621         /*
1622          * Initialize the FSLargestDataPacket value in the frame interval
1623          * register. The controller compares the value of MaxPacketSize to
1624          * this value to see if the entire packet may be sent out before
1625          * the EOF.
1626          */
1627         max_packet = ((((ohcip->ohci_frame_interval -
1628             MAX_OVERHEAD) * 6) / 7) << HCR_FRME_FSMPS_SHFT);
```

```
1630         Set_OpReg(hcr_frame_interval,
1631             (max_packet | ohcip->ohci_frame_interval));

1633         /*
1634          * Sometimes the HcFmInterval register in OHCI controller does not
1635          * maintain its value after the first write. This problem is found
1636          * on ULI M1575 South Bridge. To workaround the hardware problem,
1637          * check the value after write and retry if the last write failed.
1638          */
1639         if (ohcip->ohci_vendor_id == PCI_ULI1575_VENID &&
1640             ohcip->ohci_device_id == PCI_ULI1575_DEVID) {
1641                 ohci_frame_interval = Get_OpReg(hcr_frame_interval);
1642                 while ((ohci_frame_interval != (max_packet |
1643                     ohcip->ohci_frame_interval))) {
1644                         if (retry >= 10) {
1645                                 USB_DPRINTF_L1(PRINT_MASK_ATTA,
1646                                     ohcip->ohci_log_hdl, "Failed to program"
1647                                     " Frame Interval Register.");

1649                                 return (DDI_FAILURE);
1650                         }
1651                         retry++;
1652                         USB_DPRINTF_L2(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1653                             "ohci_init_ctlr: Failed to program Frame"
1654                             " Interval Register, retry=%d", retry);
1655                         Set_OpReg(hcr_frame_interval,
1656                             (max_packet | ohcip->ohci_frame_interval));
1657                         ohci_frame_interval = Get_OpReg(hcr_frame_interval);
1658                 }
1659         }

1661         /* Begin sending SOFs */
1662         curr_control = Get_OpReg(hcr_control);

1664         USB_DPRINTF_L4(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1665             "ohci_init_ctlr: curr_control=0x%x", curr_control);

1667         /* Set the state to operational */
1668         curr_control = (curr_control &
1669             (~HCR_CONTROL_HCFS)) | HCR_CONTROL_OPERAT;

1671         Set_OpReg(hcr_control, curr_control);

1673         ASSERT((Get_OpReg(hcr_control) &
1674             HCR_CONTROL_HCFS) == HCR_CONTROL_OPERAT);

1676         /* Set host controller soft state to operational */
1677         ohcip->ohci_hc_soft_state = OHCI_CTLR_OPERATIONAL_STATE;

1679         /* Get the number of clock ticks to wait */
1680         sof_time_wait = drv_sectohz(OHCI_MAX_SOF_TIMEWAIT);
1680         sof_time_wait = drv_usectohz(OHCI_MAX_SOF_TIMEWAIT * 1000000);

1682         /* Clear ohci_sof_flag indicating waiting for SOF interrupt */
1683         ohcip->ohci_sof_flag = B_FALSE;

1685         /* Enable the SOF interrupt */
1686         Set_OpReg(hcr_intr_enable, HCR_INTR_SOF);

1688         ASSERT(Get_OpReg(hcr_intr_enable) & HCR_INTR_SOF);

1690         (void) cv_reltimedwait(&ohcip->ohci_SOF_cv,
1691             &ohcip->ohci_int_mutex, sof_time_wait, TR_CLOCK_TICK);

1693         /* Wait for the SOF or timeout event */
1694         if (ohcip->ohci_sof_flag == B_FALSE) {
```



```
1696                 /* Set host controller soft state to error */
1697                 ohcip->ohci_hc_soft_state = OHCI_CTLR_ERROR_STATE;

1699                 USB_DPRINTF_L0(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1700                     "No SOF interrupts have been received, this USB OHCI host"
1701                     "controller is unusable");
1702                 return (DDI_FAILURE);
1703         }

1705         USB_DPRINTF_L4(PRINT_MASK_ATTA, ohcip->ohci_log_hdl,
1706             "ohci_init_ctlr: SOF's have started");

1708         return (DDI_SUCCESS);
1709 }
```

_____***unchanged_portion_omitted_***

```
7495 /*
7496  * ohci_start_timer:
7497  *
7498  * Start the ohci timer
7499  */
7500 static void
7501 ohci_start_timer(ohci_state_t   *ohcip)
7502 {
7503         USB_DPRINTF_L3(PRINT_MASK_LISTS,  ohcip->ohci_log_hdl,
7504             "ohci_start_timer: ohcip = 0x%p", (void *)ohcip);

7506         ASSERT(mutex_owned(&ohcip->ohci_int_mutex));

7508         /*
7509          * Start the global timer only if currently timer is not
7510          * running and if there are any transfers on the timeout
7511          * list. This timer will be per USB Host Controller.
7512          */
7513         if ((!ohcip->ohci_timer_id) && (ohcip->ohci_timeout_list)) {
7514                 ohcip->ohci_timer_id = timeout(ohci_xfer_timeout_handler,
7515                     (void *)ohcip, drv_sectohz(1));
7515                     (void *)ohcip, drv_usectohz(1000000));
7516         }
7517 }
```

_____***unchanged_portion_omitted_***

```
9729 /*
9730  * ohci_do_soft_reset
9731  *
9732  * Do soft reset of ohci host controller.
9733  */
9734 int
9735 ohci_do_soft_reset(ohci_state_t *ohcip)
9736 {
9737         usb_frame_number_t      before_frame_number, after_frame_number;
9738         timeout_id_t            xfer_timer_id, rh_timer_id;
9739         ohci_regs_t             *ohci_save_regs;
9740         ohci_td_t               *done_head;

9742         ASSERT(mutex_owned(&ohcip->ohci_int_mutex));

9744         /* Increment host controller error count */
9745         ohcip->ohci_hc_error++;

9747         USB_DPRINTF_L3(PRINT_MASK_INTR, ohcip->ohci_log_hdl,
9748             "ohci_do_soft_reset:"
9749             "Reset ohci host controller 0x%x", ohcip->ohci_hc_error);
```

```
9751         /*
9752          * Allocate space for saving current Host Controller
9753          * registers. Don't do any recovery if allocation
9754          * fails.
9755          */
9756         ohci_save_regs = (ohci_regs_t *)
9757             kmem_zalloc(sizeof (ohci_regs_t), KM_NOSLEEP);

9759         if (ohci_save_regs == NULL) {
9760                 USB_DPRINTF_L2(PRINT_MASK_INTR,  ohcip->ohci_log_hdl,
9761                     "ohci_do_soft_reset: kmem_zalloc failed");

9763                 return (USB_FAILURE);
9764         }

9766         /* Save current ohci registers */
9767         ohci_save_regs->hcr_control = Get_OpReg(hcr_control);
9768         ohci_save_regs->hcr_cmd_status = Get_OpReg(hcr_cmd_status);
9769         ohci_save_regs->hcr_intr_enable = Get_OpReg(hcr_intr_enable);
9770         ohci_save_regs->hcr_periodic_strt = Get_OpReg(hcr_periodic_strt);
9771         ohci_save_regs->hcr_frame_interval = Get_OpReg(hcr_frame_interval);
9772         ohci_save_regs->hcr_HCCA = Get_OpReg(hcr_HCCA);
9773         ohci_save_regs->hcr_bulk_head = Get_OpReg(hcr_bulk_head);
9774         ohci_save_regs->hcr_ctrl_head = Get_OpReg(hcr_ctrl_head);

9776         USB_DPRINTF_L4(PRINT_MASK_INTR, ohcip->ohci_log_hdl,
9777             "ohci_do_soft_reset: Save reg = 0x%p", (void *)ohci_save_regs);

9779         /* Disable all list processing and interrupts */
9780         Set_OpReg(hcr_control, (Get_OpReg(hcr_control) & ~(HCR_CONTROL_CLE |
9781             HCR_CONTROL_BLE | HCR_CONTROL_PLE | HCR_CONTROL_IE)));

9783         Set_OpReg(hcr_intr_disable, HCR_INTR_SO |
9784             HCR_INTR_WDH | HCR_INTR_RD | HCR_INTR_UE |
9785             HCR_INTR_FNO | HCR_INTR_SOF | HCR_INTR_MIE);

9787         /* Wait for few milliseconds */
9788         drv_usecwait(OHCI_TIMEWAIT);

9790         /* Root hub interrupt pipe timeout id */
9791         rh_timer_id = ohcip->ohci_root_hub.rh_intr_pipe_timer_id;

9793         /* Stop the root hub interrupt timer */
9794         if (rh_timer_id) {
9795                 ohcip->ohci_root_hub.rh_intr_pipe_timer_id = 0;
9796                 ohcip->ohci_root_hub.rh_intr_pipe_state =
9797                     OHCI_PIPE_STATE_IDLE;

9799                 mutex_exit(&ohcip->ohci_int_mutex);
9800                 (void) untimeout(rh_timer_id);
9801                 mutex_enter(&ohcip->ohci_int_mutex);
9802         }

9804         /* Transfer timeout id */
9805         xfer_timer_id = ohcip->ohci_timer_id;

9807         /* Stop the global transfer timer */
9808         if (xfer_timer_id) {
9809                 ohcip->ohci_timer_id = 0;
9810                 mutex_exit(&ohcip->ohci_int_mutex);
9811                 (void) untimeout(xfer_timer_id);
9812                 mutex_enter(&ohcip->ohci_int_mutex);
9813         }

9815         /* Process any pending HCCA DoneHead */
```



```
9816         done_head = (ohci_td_t *)(uintptr_t)
9817             (Get_HCCA(ohcip->ohci_hccap->HccaDoneHead) & HCCA_DONE_HEAD_MASK);

9819         if (ohci_check_done_head(ohcip, done_head) == USB_SUCCESS) {
9820                 /* Reset the done head to NULL */
9821                 Set_HCCA(ohcip->ohci_hccap->HccaDoneHead, NULL);

9823                 ohci_traverse_done_list(ohcip, done_head);
9824         }

9826         /* Process any pending hcr_done_head value */
9827         done_head = (ohci_td_t *)(uintptr_t)
9828             (Get_OpReg(hcr_done_head) & HCCA_DONE_HEAD_MASK);
9829         if (ohci_check_done_head(ohcip, done_head) == USB_SUCCESS) {

9831                 ohci_traverse_done_list(ohcip, done_head);
9832         }

9834         /* Do soft reset of ohci host controller */
9835         Set_OpReg(hcr_cmd_status, HCR_STATUS_RESET);

9837         USB_DPRINTF_L3(PRINT_MASK_INTR, ohcip->ohci_log_hdl,
9838             "ohci_do_soft_reset: Reset in progress");

9840         /* Wait for reset to complete */
9841         drv_usecwait(OHCI_RESET_TIMEWAIT);

9843         /* Reset HCCA HcFrameNumber */
9844         Set_HCCA(ohcip->ohci_hccap->HccaFrameNo, 0x00000000);

9846         /*
9847          * Restore previous saved HC register value
9848          * into the current HC registers.
9849          */
9850         Set_OpReg(hcr_periodic_strt, (uint32_t)
9851             ohci_save_regs->hcr_periodic_strt);

9853         Set_OpReg(hcr_frame_interval, (uint32_t)
9854             ohci_save_regs->hcr_frame_interval);

9856         Set_OpReg(hcr_done_head, 0x0);

9858         Set_OpReg(hcr_bulk_curr, 0x0);

9860         Set_OpReg(hcr_bulk_head, (uint32_t)
9861             ohci_save_regs->hcr_bulk_head);

9863         Set_OpReg(hcr_ctrl_curr, 0x0);

9865         Set_OpReg(hcr_ctrl_head, (uint32_t)
9866             ohci_save_regs->hcr_ctrl_head);

9868         Set_OpReg(hcr_periodic_curr, 0x0);

9870         Set_OpReg(hcr_HCCA, (uint32_t)
9871             ohci_save_regs->hcr_HCCA);

9873         Set_OpReg(hcr_intr_status, 0x0);

9875         /*
9876          * Set HcInterruptEnable to enable all interrupts except
9877          * Root Hub Status change interrupt.
9878          */
9879         Set_OpReg(hcr_intr_enable,
9880             HCR_INTR_SO | HCR_INTR_WDH | HCR_INTR_RD | HCR_INTR_UE |
9881             HCR_INTR_FNO | HCR_INTR_SOF | HCR_INTR_MIE);
```

```
9883         /* Start Control and Bulk list processing */
9884         Set_OpReg(hcr_cmd_status, (HCR_STATUS_CLF | HCR_STATUS_BLF));

9886         /*
9887          * Start up Control, Bulk, Periodic and Isochronous lists
9888          * processing.
9889          */
9890         Set_OpReg(hcr_control, (uint32_t)
9891             (ohci_save_regs->hcr_control & (~HCR_CONTROL_HCFS)));

9893         /*
9894          * Deallocate the space that allocated for saving
9895          * HC registers.
9896          */
9897         kmem_free((void *) ohci_save_regs, sizeof (ohci_regs_t));

9899         /* Resume the host controller */
9900         Set_OpReg(hcr_control, ((Get_OpReg(hcr_control) &
9901             (~HCR_CONTROL_HCFS)) | HCR_CONTROL_RESUME));

9903         /* Wait for resume to complete */
9904         drv_usecwait(OHCI_RESUME_TIMEWAIT);

9906         /* Set the Host Controller Functional State to Operational */
9907         Set_OpReg(hcr_control, ((Get_OpReg(hcr_control) &
9908             (~HCR_CONTROL_HCFS)) | HCR_CONTROL_OPERAT));

9910         /* Wait 10ms for HC to start sending SOF */
9911         drv_usecwait(OHCI_TIMEWAIT);

9913         /*
9914          * Get the current usb frame number before waiting for few
9915          * milliseconds.
9916          */
9917         before_frame_number = ohci_get_current_frame_number(ohcip);

9919         /* Wait for few milliseconds */
9920         drv_usecwait(OHCI_TIMEWAIT);

9922         /*
9923          * Get the current usb frame number after waiting for few
9924          * milliseconds.
9925          */
9926         after_frame_number = ohci_get_current_frame_number(ohcip);

9928         USB_DPRINTF_L3(PRINT_MASK_INTR, ohcip->ohci_log_hdl,
9929             "ohci_do_soft_reset: Before Frm No 0x%llx After Frm No 0x%llx",
9930             (unsigned long long)before_frame_number,
9931             (unsigned long long)after_frame_number);

9933         if (after_frame_number <= before_frame_number) {

9935                 USB_DPRINTF_L2(PRINT_MASK_INTR, ohcip->ohci_log_hdl,
9936                     "ohci_do_soft_reset: Soft reset failed");

9938                 return (USB_FAILURE);
9939         }

9941         /* Start the timer for the root hub interrupt pipe polling */
9942         if (rh_timer_id) {
9943                 ohcip->ohci_root_hub.rh_intr_pipe_timer_id =
9944                     timeout(ohci_handle_root_hub_status_change,
9945                     (void *)ohcip, drv_usectohz(OHCI_RH_POLL_TIME));

9947                 ohcip->ohci_root_hub.
```



```
9948                         rh_intr_pipe_state = OHCI_PIPE_STATE_ACTIVE;
9949         }

9951         /* Start the global timer */
9952         if (xfer_timer_id) {
9953                 ohcip->ohci_timer_id = timeout(ohci_xfer_timeout_handler,
9954                     (void *)ohcip, drv_sectohz(1));
9954                     (void *)ohcip, drv_usectohz(1000000));
9955         }

9957         return (USB_SUCCESS);
9958 }
______unchanged_portion_omitted_


10182 /*
10183  * ohci_wait_for_sof:
10184  *
10185  * Wait for couple of SOF interrupts
10186  */
10187 static int
10188 ohci_wait_for_sof(ohci_state_t  *ohcip)
10189 {
10190         usb_frame_number_t      before_frame_number, after_frame_number;
10191         clock_t                 sof_time_wait;
10192         int                     rval, sof_wait_count;

10194         USB_DPRINTF_L4(PRINT_MASK_LISTS, ohcip->ohci_log_hdl,
10195             "ohci_wait_for_sof");

10197         ASSERT(mutex_owned(&ohcip->ohci_int_mutex));

10199         rval = ohci_state_is_operational(ohcip);

10201         if (rval != USB_SUCCESS) {

10203                 return (rval);
10204         }

10206         /* Get the number of clock ticks to wait */
10207         sof_time_wait = drv_sectohz(OHCI_MAX_SOF_TIMEWAIT);
10207         sof_time_wait = drv_usectohz(OHCI_MAX_SOF_TIMEWAIT * 1000000);

10209         sof_wait_count = 0;

10211         /*
10212          * Get the current usb frame number before waiting for the
10213          * SOF interrupt event.
10214          */
10215         before_frame_number = ohci_get_current_frame_number(ohcip);

10217         while (sof_wait_count < MAX_SOF_WAIT_COUNT) {
10218                 /* Enable the SOF interrupt */
10219                 Set_OpReg(hcr_intr_enable, HCR_INTR_SOF);

10221                 ASSERT(Get_OpReg(hcr_intr_enable) & HCR_INTR_SOF);

10223                 /* Wait for the SOF or timeout event */
10224                 rval = cv_reltimedwait(&ohcip->ohci_SOF_cv,
10225                     &ohcip->ohci_int_mutex, sof_time_wait, TR_CLOCK_TICK);

10227                 /*
10228                  * Get the current usb frame number after woken up either
10229                  * from SOF interrupt or timer expired event.
10230                  */
10231                 after_frame_number = ohci_get_current_frame_number(ohcip);
```

```
10233                 USB_DPRINTF_L3(PRINT_MASK_LISTS, ohcip->ohci_log_hdl,
10234                     "ohci_wait_for_sof: before 0x%llx, after 0x%llx",
10235                     (unsigned long long)before_frame_number,
10236                     (unsigned long long)after_frame_number);

10238                 /*
10239                  * Return failure, if we are woken up becuase of timer expired
10240                  * event and if usb frame number has not been changed.
10241                  */
10242                 if ((rval == -1) &&
10243                     (after_frame_number <= before_frame_number)) {

10245                         if ((ohci_do_soft_reset(ohcip)) != USB_SUCCESS) {

10247                                 USB_DPRINTF_L0(PRINT_MASK_LISTS,
10248                                     ohcip->ohci_log_hdl, "No SOF interrupts");

10250                                 /* Set host controller soft state to error */
10251                                 ohcip->ohci_hc_soft_state =
10252                                     OHCI_CTLR_ERROR_STATE;

10254                                 return (USB_FAILURE);
10255                         }

10257                         /* Get new usb frame number */
10258                         after_frame_number = before_frame_number =
10259                             ohci_get_current_frame_number(ohcip);
10260                 }

10262                 ASSERT(after_frame_number >= before_frame_number);

10264                 before_frame_number = after_frame_number;
10265                 sof_wait_count++;
10266         }

10268         return (USB_SUCCESS);
10269 }
```

unchanged_portion_omitted_

```
10402 /*
10403  * ohci_wait_for_transfers_completion:
10404  *
10405  * Wait for processing all completed transfers and to send results
10406  * to upstream.
10407  */
10408 static void
10409 ohci_wait_for_transfers_completion(
10410         ohci_state_t            *ohcip,
10411         ohci_pipe_private_t     *pp)
10412 {
10413         ohci_trans_wrapper_t    *head_tw = pp->pp_tw_head;
10414         ohci_trans_wrapper_t    *next_tw;
10415         ohci_td_t               *tailp, *headp, *nextp;
10416         ohci_td_t               *head_td, *next_td;
10417         ohci_ed_t               *ept = pp->pp_ept;
10418         int                     rval;

10420         USB_DPRINTF_L4(PRINT_MASK_LISTS, ohcip->ohci_log_hdl,
10421             "ohci_wait_for_transfers_completion: pp = 0x%p", (void *)pp);

10423         ASSERT(mutex_owned(&ohcip->ohci_int_mutex));

10425         headp = (ohci_td_t *)(ohci_td_iommu_to_cpu(ohcip,
10426             Get_ED(ept->hced_headp) & (uint32_t)HC_EPT_TD_HEAD));
```



```
10428         tailp = (ohci_td_t *)(ohci_td_iommu_to_cpu(ohcip,
10429             Get_ED(ept->hced_tailp) & (uint32_t)HC_EPT_TD_TAIL));

10431         rval = ohci_state_is_operational(ohcip);

10433         if (rval != USB_SUCCESS) {

10435                 return;
10436         }

10438         pp->pp_count_done_tds = 0;

10440         /* Process the transfer wrappers for this pipe */
10441         next_tw = head_tw;
10442         while (next_tw) {
10443                 head_td = (ohci_td_t *)next_tw->tw_hctd_head;
10444                 next_td = head_td;

10446                 if (head_td) {
10447                         /*
10448                          * Walk through each TD for this transfer
10449                          * wrapper. If a TD still exists, then it
10450                          * is currently on the done list.
10451                          */
10452                         while (next_td) {

10454                                 nextp = headp;

10456                                 while (nextp != tailp) {

10458                                         /* TD is on the ED */
10459                                         if (nextp == next_td) {
10460                                                 break;
10461                                         }

10463                                         nextp = (ohci_td_t *)
10464                                             (ohci_td_iommu_to_cpu(ohcip,
10465                                             (Get_TD(nextp->hctd_next_td) &
10466                                             HC_EPT_TD_TAIL)));
10467                                 }

10469                                 if (nextp == tailp) {
10470                                         pp->pp_count_done_tds++;
10471                                 }

10473                                 next_td = ohci_td_iommu_to_cpu(ohcip,
10474                                     Get_TD(next_td->hctd_tw_next_td));
10475                         }
10476                 }

10478                 next_tw = next_tw->tw_next;
10479         }

10481         USB_DPRINTF_L3(PRINT_MASK_LISTS, ohcip->ohci_log_hdl,
10482             "ohci_wait_for_transfers_completion: count_done_tds = 0x%x",
10483             pp->pp_count_done_tds);

10485         if (!pp->pp_count_done_tds) {

10487                 return;
10488         }

10490         (void) cv_reltimedwait(&pp->pp_xfer_cmpl_cv, &ohcip->ohci_int_mutex,
10491             drv_sectohz(OHCI_XFER_CMPL_TIMEWAIT), TR_CLOCK_TICK);
10491             drv_usectohz(OHCI_XFER_CMPL_TIMEWAIT * 1000000), TR_CLOCK_TICK);
```

```
10493         if (pp->pp_count_done_tds) {

10495                 USB_DPRINTF_L2(PRINT_MASK_LISTS, ohcip->ohci_log_hdl,
10496                     "ohci_wait_for_transfers_completion: No transfers "
10497                     "completion confirmation received for 0x%x requests",
10498                     pp->pp_count_done_tds);
10499         }
10500 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   48316 Wed Aug 19 07:25:18 2015
new/usr/src/uts/common/io/usb/hcd/uhci/uhci.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 234 /*
 235  * Host Controller Driver (HCD) Auto configuration entry points
 236  */

 238 /*
 239  * Function Name  :  uhci_attach:
 240  * Description    :  Attach entry point - called by the Kernel.
 241  *                   Allocates of per controller data structure.
 242  *                   Initializes the controller.
 243  * Output         :  DDI_SUCCESS / DDI_FAILURE
 244  */
 245 static int
 246 uhci_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 247 {
 248         int                             instance, polled;
 249         int                             i, intr_types;
 250         uhci_state_t                    *uhcip = NULL;
 251         usba_hcdi_register_args_t       hcdi_args;

 253         USB_DPRINTF_L4(PRINT_MASK_ATTA, NULL, "uhci_attach:");

 255         switch (cmd) {
 256         case DDI_ATTACH:
 257                 break;
 258         case DDI_RESUME:
 259                 uhcip = uhci_obtain_state(dip);

 261                 return (uhci_cpr_resume(uhcip));
 262         default:

 264                 return (DDI_FAILURE);
 265         }

 267         /* Get the instance and create soft state */
 268         instance = ddi_get_instance(dip);

 270         /* Allocate the soft state structure for this instance of the driver */
 271         if (ddi_soft_state_zalloc(uhci_statep, instance) != 0) {

 273                 return (DDI_FAILURE);
 274         }

 276         if ((uhcip = ddi_get_soft_state(uhci_statep, instance)) == NULL) {

 278                 return (DDI_FAILURE);
 279         }

 281         uhcip->uhci_log_hdl = usb_alloc_log_hdl(dip, "uhci", &uhci_errlevel,
 282             &uhci_errmask, &uhci_instance_debug, 0);

 284         /* Set host controller soft state to initialization */
 285         uhcip->uhci_hc_soft_state = UHCI_CTLR_INIT_STATE;

 287         /* Save the dip and instance */
 288         uhcip->uhci_dip         = dip;
 289         uhcip->uhci_instance    = instance;

 291         polled = uhci_is_polled(dip);
 292         if (polled)
```



```
 294                 goto skip_intr;

 296         /* Get supported interrupt types */
 297         if (ddi_intr_get_supported_types(uhcip->uhci_dip,
 298             &intr_types) != DDI_SUCCESS) {
 299                 USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 300                     "uhci_attach: ddi_intr_get_supported_types failed");

 302                 usb_free_log_hdl(uhcip->uhci_log_hdl);
 303                 ddi_soft_state_free(uhci_statep, instance);

 305                 return (DDI_FAILURE);
 306         }

 308         USB_DPRINTF_L3(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 309             "uhci_attach: supported interrupt types 0x%x", intr_types);

 311         if ((intr_types & DDI_INTR_TYPE_MSI) && uhci_enable_msi) {
 312                 if (uhci_add_intrs(uhcip, DDI_INTR_TYPE_MSI)
 313                     != DDI_SUCCESS) {
 314                         USB_DPRINTF_L4(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 315                             "uhci_attach: MSI registration failed, "
 316                             "trying FIXED interrupt \n");
 317                 } else {
 318                         USB_DPRINTF_L4(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 319                             "uhci_attach: Using MSI interrupt type\n");

 321                         uhcip->uhci_intr_type = DDI_INTR_TYPE_MSI;
 322                 }
 323         }

 325         if (!(uhcip->uhci_htable) && (intr_types & DDI_INTR_TYPE_FIXED)) {
 326                 if (uhci_add_intrs(uhcip, DDI_INTR_TYPE_FIXED)
 327                     != DDI_SUCCESS) {
 328                         USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 329                             "uhci_attach: FIXED interrupt registration "
 330                             "failed\n");

 332                         usb_free_log_hdl(uhcip->uhci_log_hdl);
 333                         ddi_soft_state_free(uhci_statep, instance);

 335                         return (DDI_FAILURE);
 336                 }

 338                 USB_DPRINTF_L4(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
 339                     "uhci_attach: Using FIXED interrupt type\n");

 341                 uhcip->uhci_intr_type = DDI_INTR_TYPE_FIXED;
 342         }

 344 skip_intr:
 345         /* Semaphore to serialize opens and closes */
 346         sema_init(&uhcip->uhci_ocsem, 1, NULL, SEMA_DRIVER, NULL);

 348         /* Create prototype condition variable */
 349         cv_init(&uhcip->uhci_cv_SOF, NULL, CV_DRIVER, NULL);

 351         /* Initialize the DMA attributes */
 352         uhci_set_dma_attributes(uhcip);

 354         /* Initialize the kstat structures */
 355         uhci_create_stats(uhcip);

 357         /* Create the td and ed pools */
 358         if (uhci_allocate_pools(uhcip) != USB_SUCCESS) {
```

```
360                 goto fail;
361         }

363         /* Map the registers */
364         if (uhci_map_regs(uhcip) != USB_SUCCESS) {

366                 goto fail;
367         }

369         /* Enable all interrupts */
370         if (polled) {
371                 extern pri_t maxclsyspri;

373                 USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
374                     "uhci_attach: running in simulated polled mode.");

376                 /* create thread to poll */
377                 (void) thread_create(NULL, 0, uhci_poll_intr, uhcip, 0, &p0,
378                     TS_RUN, maxclsyspri);
379         } else if (uhcip->uhci_intr_cap & DDI_INTR_FLAG_BLOCK) {
380                 /* Call ddi_intr_block_enable() for MSI interrupts */
381                 (void) ddi_intr_block_enable(uhcip->uhci_htable,
382                     uhcip->uhci_intr_cnt);
383         } else {
384                 /* Call ddi_intr_enable for MSI or FIXED interrupts */
385                 for (i = 0; i < uhcip->uhci_intr_cnt; i++)
386                         (void) ddi_intr_enable(uhcip->uhci_htable[i]);
387         }


390         /* Initialize the controller */
391         if (uhci_init_ctlr(uhcip) != USB_SUCCESS) {

393                 goto fail;
394         }

396         /*
397          * At this point, the hardware will be okay.
398          * Initialize the usba_hcdi structure
399          */
400         uhcip->uhci_hcdi_ops = uhci_alloc_hcdi_ops(uhcip);

402         /*
403          * Make this HCD instance known to USBA
404          * (dma_attr must be passed for USBA busctl's)
405          */
406         hcdi_args.usba_hcdi_register_version = HCDI_REGISTER_VERSION;
407         hcdi_args.usba_hcdi_register_dip = dip;
408         hcdi_args.usba_hcdi_register_ops = uhcip->uhci_hcdi_ops;
409         hcdi_args.usba_hcdi_register_dma_attr = &uhcip->uhci_dma_attr;
410         hcdi_args.usba_hcdi_register_iblock_cookie =
411             (ddi_iblock_cookie_t)(uintptr_t)uhcip->uhci_intr_pri;

413         if (usba_hcdi_register(&hcdi_args, 0) != USB_SUCCESS) {

415                 goto fail;
416         }

418 #ifndef __sparc
419         /*
420          * On NCR system,  the driver seen  failure of some commands
421          * while booting. This delay mysteriously solved the problem.
422          */
423         delay(drv_sectohz(uhci_attach_wait));
423         delay(drv_usectohz(uhci_attach_wait*1000000));
```



```
424 #endif

426         /*
427          * Create another timeout handler to check whether any
428          * control/bulk/interrupt commands failed.
429          * This gets called every second.
430          */
431         uhcip->uhci_cmd_timeout_id = timeout(uhci_cmd_timeout_hdlr,
432             (void *)uhcip, UHCI_ONE_SECOND);

434         mutex_enter(&uhcip->uhci_int_mutex);

436         /*
437          * Set HcInterruptEnable to enable all interrupts except Root
438          * Hub Status change and SOF interrupts.
439          */
440         Set_OpReg16(USBINTR, ENABLE_ALL_INTRS);

442         /* Test the SOF interrupt */
443         if (uhci_wait_for_sof(uhcip) != USB_SUCCESS) {
444                 USB_DPRINTF_L0(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
445                     "No SOF interrupts have been received, this USB UHCI host"
446                     " controller is unusable");
447                 mutex_exit(&uhcip->uhci_int_mutex);

449                 goto fail;
450         }

452         mutex_exit(&uhcip->uhci_int_mutex);

454         /* This should be the last step which might fail during attaching */
455         if (uhci_init_root_hub(uhcip) != USB_SUCCESS) {

457                 goto fail;
458         }

460         /* Display information in the banner */
461         ddi_report_dev(dip);

463         USB_DPRINTF_L4(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
464             "uhci_attach successful");

466         return (DDI_SUCCESS);

468 fail:
469         USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
470             "failed to attach");

472         (void) uhci_cleanup(uhcip);

474         return (DDI_FAILURE);
475 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  159872 Wed Aug 19 07:25:19 2015
new/usr/src/uts/common/io/usb/scsa2usb/scsa2usb.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


1300 /*
1301  * scsa2usb_cleanup:
1302  *      cleanup whatever attach has setup
1303  */
1304 static int
1305 scsa2usb_cleanup(dev_info_t *dip, scsa2usb_state_t *scsa2usbp)
1306 {
1307         int             rval, i;
1308         scsa2usb_power_t *pm;
1309         uint_t          lun;

1311         USB_DPRINTF_L4(DPRINT_MASK_SCSA, scsa2usbp->scsa2usb_log_handle,
1312             "scsa2usb_cleanup:");

1314         /* wait till the work thread is done */
1315         mutex_enter(&scsa2usbp->scsa2usb_mutex);
1316         for (i = 0; i < SCSA2USB_DRAIN_TIMEOUT; i++) {
1317                 if (scsa2usbp->scsa2usb_work_thread_id == NULL) {

1319                         break;
1320                 }
1321                 mutex_exit(&scsa2usbp->scsa2usb_mutex);
1322                 delay(drv_sectohz(1));
1322                 delay(drv_usectohz(1000000));
1323                 mutex_enter(&scsa2usbp->scsa2usb_mutex);
1324         }
1325         mutex_exit(&scsa2usbp->scsa2usb_mutex);

1327         if (i >= SCSA2USB_DRAIN_TIMEOUT) {

1329                 return (USB_FAILURE);
1330         }

1332         /*
1333          * Disable the event callbacks first, after this point, event
1334          * callbacks will never get called. Note we shouldn't hold
1335          * mutex while unregistering events because there may be a
1336          * competing event callback thread. Event callbacks are done
1337          * with ndi mutex held and this can cause a potential deadlock.
1338          */
1339         usb_unregister_event_cbs(scsa2usbp->scsa2usb_dip, &scsa2usb_events);

1341         if (scsa2usbp->scsa2usb_flags & SCSA2USB_FLAGS_LOCKS_INIT) {
1342                 /*
1343                  * if a waitQ exists, get rid of it before destroying it
1344                  */
1345                 for (lun = 0; lun < SCSA2USB_MAX_LUNS; lun++) {
1346                         scsa2usb_flush_waitQ(scsa2usbp, lun, CMD_TRAN_ERR);
1347                         usba_destroy_list(&scsa2usbp->scsa2usb_waitQ[lun]);
1348                 }

1350                 mutex_enter(&scsa2usbp->scsa2usb_mutex);
1351                 if (scsa2usbp->scsa2usb_flags &
1352                     SCSA2USB_FLAGS_HBA_ATTACH_SETUP) {
1353                         (void) scsi_hba_detach(dip);
1354                         scsi_hba_tran_free(scsa2usbp->scsa2usb_tran);
1355                 }
```



```
1357                 if (scsa2usbp->scsa2usb_flags &
1358                     SCSA2USB_FLAGS_PIPES_OPENED) {
1359                         scsa2usb_close_usb_pipes(scsa2usbp);
1360                 }

1362                 /* Lower the power */
1363                 pm = scsa2usbp->scsa2usb_pm;

1365                 if (pm && (scsa2usbp->scsa2usb_dev_state !=
1366                     USB_DEV_DISCONNECTED)) {
1367                         if (pm->scsa2usb_wakeup_enabled) {
1368                                 mutex_exit(&scsa2usbp->scsa2usb_mutex);
1369                                 (void) pm_raise_power(dip, 0,
1370                                     USB_DEV_OS_FULL_PWR);

1372                                 if ((rval = usb_handle_remote_wakeup(dip,
1373                                     USB_REMOTE_WAKEUP_DISABLE)) !=
1374                                     USB_SUCCESS) {
1375                                         USB_DPRINTF_L2(DPRINT_MASK_SCSA,
1376                                             scsa2usbp->scsa2usb_log_handle,
1377                                             "disable remote wakeup failed "
1378                                             "(%d)", rval);
1379                                 }
1380                         } else {
1381                                 mutex_exit(&scsa2usbp->scsa2usb_mutex);
1382                         }

1384                         (void) pm_lower_power(dip, 0, USB_DEV_OS_PWR_OFF);

1386                         mutex_enter(&scsa2usbp->scsa2usb_mutex);
1387                 }

1389                 if (pm) {
1390                         kmem_free(pm, sizeof (scsa2usb_power_t));
1391                 }

1393                 if (scsa2usbp->scsa2usb_override_str) {
1394                         kmem_free(scsa2usbp->scsa2usb_override_str,
1395                             strlen(scsa2usbp->scsa2usb_override_str) + 1);
1396                         scsa2usbp->scsa2usb_override_str = NULL;
1397                 }

1399                 /* remove the minor nodes */
1400                 ddi_remove_minor_node(dip, NULL);

1402                 /* Cancel the registered panic callback */
1403                 scsa2usb_panic_callb_fini(scsa2usbp);

1405                 mutex_exit(&scsa2usbp->scsa2usb_mutex);

1407                 mutex_destroy(&scsa2usbp->scsa2usb_mutex);
1408                 cv_destroy(&scsa2usbp->scsa2usb_transport_busy_cv);
1409         }

1411         usb_client_detach(scsa2usbp->scsa2usb_dip,
1412             scsa2usbp->scsa2usb_dev_data);

1414         if (scsa2usbp->scsa2usb_ugen_hdl) {
1415                 (void) usb_ugen_detach(scsa2usbp->scsa2usb_ugen_hdl,
1416                     DDI_DETACH);
1417                 usb_ugen_release_hdl(scsa2usbp->scsa2usb_ugen_hdl);
1418         }

1420         usb_free_log_hdl(scsa2usbp->scsa2usb_log_handle);

1422         ddi_prop_remove_all(dip);
```

```
1424         ddi_soft_state_free(scsa2usb_statep, ddi_get_instance(dip));

1426         return (USB_SUCCESS);
1427 }
```

_____***unchanged_portion_omitted_***

```
5422 /*
5423  * scsa2usb_disconnect_event_cb:
5424  *      callback for disconnect events
5425  */
5426 static int
5427 scsa2usb_disconnect_event_cb(dev_info_t *dip)
5428 {
5429         scsa2usb_state_t *scsa2usbp =
5430             ddi_get_soft_state(scsa2usb_statep, ddi_get_instance(dip));
5431         dev_info_t      *cdip;
5432         int             circ, i;
5433         int             rval = USB_SUCCESS;

5435         ASSERT(scsa2usbp != NULL);

5437         USB_DPRINTF_L4(DPRINT_MASK_SCSA, scsa2usbp->scsa2usb_log_handle,
5438             "scsa2usb_disconnect_event_cb: dip = 0x%p", (void *)dip);

5440         mutex_enter(&scsa2usbp->scsa2usb_mutex);
5441         scsa2usbp->scsa2usb_dev_state = USB_DEV_DISCONNECTED;

5443         /*
5444          * wait till the work thread is done, carry on regardless
5445          * if not.
5446          */
5447         for (i = 0; i < SCSA2USB_DRAIN_TIMEOUT; i++) {
5448                 if ((scsa2usbp->scsa2usb_work_thread_id == NULL) &&
5449                     (scsa2usbp->scsa2usb_cur_pkt == NULL) &&
5450                     (scsa2usb_all_waitQs_empty(scsa2usbp) ==
5451                     USB_SUCCESS)) {

5453                         break;
5454                 }
5455                 mutex_exit(&scsa2usbp->scsa2usb_mutex);
5456                 delay(drv_sectohz(1));
5456                 delay(drv_usectohz(1000000));
5457                 mutex_enter(&scsa2usbp->scsa2usb_mutex);
5458         }
5459         mutex_exit(&scsa2usbp->scsa2usb_mutex);

5461         ndi_devi_enter(dip, &circ);
5462         for (cdip = ddi_get_child(dip); cdip; ) {
5463                 dev_info_t *next = ddi_get_next_sibling(cdip);

5465                 mutex_enter(&DEVI(cdip)->devi_lock);
5466                 DEVI_SET_DEVICE_REMOVED(cdip);
5467                 mutex_exit(&DEVI(cdip)->devi_lock);

5469                 cdip = next;
5470         }
5471         ndi_devi_exit(dip, circ);

5473         if (scsa2usbp->scsa2usb_ugen_hdl) {
5474                 rval = usb_ugen_disconnect_ev_cb(
5475                     scsa2usbp->scsa2usb_ugen_hdl);
5476         }

5478         return (rval);
```



```
5479 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  235051 Wed Aug 19 07:25:19 2015
new/usr/src/uts/common/io/usb/usba/hubdi.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

8601 /*
8602  * hubd_wait_for_hotplug_exit:
8603  *      Waiting for the exit of the running hotplug thread or ioctl thread.
8604  */
8605 static int
8606 hubd_wait_for_hotplug_exit(hubd_t *hubd)
8607 {
8608         clock_t         until = drv_sectohz(1);
8608         clock_t         until = drv_usectohz(1000000);
8609         int             rval;

8611         ASSERT(mutex_owned(HUBD_MUTEX(hubd)));

8613         if (hubd->h_hotplug_thread) {
8614                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8615                     "waiting for hubd hotplug thread exit");
8616                 rval = cv_reltimedwait(&hubd->h_cv_hotplug_dev,
8617                     &hubd->h_mutex, until, TR_CLOCK_TICK);

8619                 if ((rval <= 0) && (hubd->h_hotplug_thread)) {

8621                         return (USB_FAILURE);
8622                 }
8623         }

8625         return (USB_SUCCESS);
8626 }

8628 /*
8629  * hubd_reset_thread:
8630  *      handles the "USB_RESET_LVL_REATTACH" reset of usb device.
8631  *
8632  *      - delete the child (force detaching the device and its children)
8633  *      - reset the corresponding parent hub port
8634  *      - create the child (force re-attaching the device and its children)
8635  */
8636 static void
8637 hubd_reset_thread(void *arg)
8638 {
8639         hubd_reset_arg_t *hd_arg = (hubd_reset_arg_t *)arg;
8640         hubd_t          *hubd = hd_arg->hubd;
8641         uint16_t        reset_port = hd_arg->reset_port;
8642         uint16_t        status, change;
8643         hub_power_t     *hubpm;
8644         dev_info_t      *hdip = hubd->h_dip;
8645         dev_info_t      *rh_dip = hubd->h_usba_device->usb_root_hub_dip;
8646         dev_info_t      *child_dip;
8647         boolean_t       online_child = B_FALSE;
8648         int             prh_circ, rh_circ, circ, devinst;
8649         char            *devname;
8650         int             i = 0;
8651         int             rval = USB_FAILURE;

8653         USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8654             "hubd_reset_thread:  started, hubd_reset_port = 0x%x", reset_port);

8656         kmem_free(arg, sizeof (hubd_reset_arg_t));

8658         mutex_enter(HUBD_MUTEX(hubd));
```



```
8660         child_dip = hubd->h_children_dips[reset_port];
8661         ASSERT(child_dip != NULL);

8663         devname = (char *)ddi_driver_name(child_dip);
8664         devinst = ddi_get_instance(child_dip);

8666         /* if our bus power entry point is active, quit the reset */
8667         if (hubd->h_bus_pwr) {
8668                 USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8669                     "%s%d is under bus power management, cannot be reset. "
8670                     "Please disconnect and reconnect this device.",
8671                     devname, devinst);

8673                 goto Fail;
8674         }

8676         if (hubd_wait_for_hotplug_exit(hubd) == USB_FAILURE) {
8677                 /* we got woken up because of a timeout */
8678                 USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG,
8679                     hubd->h_log_handle, "Time out when resetting the device"
8680                     " %s%d. Please disconnect and reconnect this device.",
8681                     devname, devinst);

8683                 goto Fail;
8684         }

8686         hubd->h_hotplug_thread++;

8688         /* is this the root hub? */
8689         if ((hdip == rh_dip) &&
8690             (hubd->h_dev_state == USB_DEV_PWRED_DOWN)) {
8691                 hubpm = hubd->h_hubpm;

8693                 /* mark the root hub as full power */
8694                 hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
8695                 hubpm->hubp_time_at_full_power = gethrtime();
8696                 mutex_exit(HUBD_MUTEX(hubd));

8698                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8699                     "hubd_reset_thread: call pm_power_has_changed");

8701                 (void) pm_power_has_changed(hdip, 0,
8702                     USB_DEV_OS_FULL_PWR);

8704                 mutex_enter(HUBD_MUTEX(hubd));
8705                 hubd->h_dev_state = USB_DEV_ONLINE;
8706         }

8708         mutex_exit(HUBD_MUTEX(hubd));

8710         /*
8711          * this ensures one reset activity per system at a time.
8712          * we enter the parent PCI node to have this serialization.
8713          * this also excludes ioctls and deathrow thread
8714          */
8715         ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8716         ndi_devi_enter(rh_dip, &rh_circ);

8718         /* exclude other threads */
8719         ndi_devi_enter(hdip, &circ);
8720         mutex_enter(HUBD_MUTEX(hubd));

8722         /*
8723          * We need to make sure that the child is still online for a hotplug
8724          * thread could have inserted which detached the child.
```

```
8725          */
8726         if (hubd->h_children_dips[reset_port]) {
8727                 mutex_exit(HUBD_MUTEX(hubd));
8728                 /* First disconnect the device */
8729                 hubd_post_event(hubd, reset_port, USBA_EVENT_TAG_HOT_REMOVAL);

8731                 /* delete cached dv_node's but drop locks first */
8732                 ndi_devi_exit(hdip, circ);
8733                 ndi_devi_exit(rh_dip, rh_circ);
8734                 ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8736                 (void) devfs_clean(rh_dip, NULL, DV_CLEAN_FORCE);

8738                 /*
8739                  * workaround only for storage device. When it's able to force
8740                  * detach a driver, this code can be removed safely.
8741                  *
8742                  * If we're to reset storage device and the device is used, we
8743                  * will wait at most extra 20s for applications to exit and
8744                  * close the device. This is especially useful for HAL-based
8745                  * applications.
8746                  */
8747                 if ((strcmp(devname, "scsa2usb") == 0) &&
8748                     DEVI(child_dip)->devi_ref != 0) {
8749                         while (i++ < hubdi_reset_delay) {
8750                                 mutex_enter(HUBD_MUTEX(hubd));
8751                                 rval = hubd_delete_child(hubd, reset_port,
8752                                     NDI_DEVI_REMOVE, B_FALSE);
8753                                 mutex_exit(HUBD_MUTEX(hubd));
8754                                 if (rval == USB_SUCCESS)
8755                                         break;

8757                                 delay(drv_sectohz(1));
8757                                 delay(drv_usectohz(1000000)); /* 1s */
8758                         }
8759                 }

8761                 ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8762                 ndi_devi_enter(rh_dip, &rh_circ);
8763                 ndi_devi_enter(hdip, &circ);

8765                 mutex_enter(HUBD_MUTEX(hubd));

8767                 /* Then force detaching the device */
8768                 if ((rval != USB_SUCCESS) && (hubd_delete_child(hubd,
8769                     reset_port, NDI_DEVI_REMOVE, B_FALSE) != USB_SUCCESS)) {
8770                         USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8771                             "%s%d cannot be reset due to other applications "
8772                             "are using it, please first close these "
8773                             "applications, then disconnect and reconnect"
8774                             "the device.", devname, devinst);

8776                         mutex_exit(HUBD_MUTEX(hubd));
8777                         /* post a re-connect event */
8778                         hubd_post_event(hubd, reset_port,
8779                             USBA_EVENT_TAG_HOT_INSERTION);
8780                         mutex_enter(HUBD_MUTEX(hubd));
8781                 } else {
8782                         (void) hubd_determine_port_status(hubd, reset_port,
8783                             &status, &change, HUBD_ACK_ALL_CHANGES);

8785                         /* Reset the parent hubd port and create new child */
8786                         if (status & PORT_STATUS_CCS) {
8787                                 online_child |= (hubd_handle_port_connect(hubd,
8788                                     reset_port) == USB_SUCCESS);
8789                         }
```



```
8790                 }
8791         }

8793         /* release locks so we can do a devfs_clean */
8794         mutex_exit(HUBD_MUTEX(hubd));

8796         /* delete cached dv_node's but drop locks first */
8797         ndi_devi_exit(hdip, circ);
8798         ndi_devi_exit(rh_dip, rh_circ);
8799         ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8801         (void) devfs_clean(rh_dip, NULL, 0);

8803         /* now check if any children need onlining */
8804         if (online_child) {
8805                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8806                     "hubd_reset_thread: onlining children");

8808                 (void) ndi_devi_online(hubd->h_dip, 0);
8809         }

8811         mutex_enter(HUBD_MUTEX(hubd));

8813         /* allow hotplug thread now */
8814         hubd->h_hotplug_thread--;
8815 Fail:
8816         hubd_start_polling(hubd, 0);

8818         /* mark this device as idle */
8819         (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);

8821         USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8822             "hubd_reset_thread: exit, %d", hubd->h_hotplug_thread);

8824         hubd->h_reset_port[reset_port] = B_FALSE;

8826         mutex_exit(HUBD_MUTEX(hubd));

8828         ndi_rele_devi(hdip);
8829 }
______unchanged_portion_omitted_
```

```
**********************************************************
   27207 Wed Aug 19 07:25:19 2015
new/usr/src/uts/common/io/vioblk/vioblk.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 457 static int
 458 vioblk_devid_init(void *arg, dev_info_t *devinfo, ddi_devid_t *devid)
 459 {
 460         struct vioblk_softc *sc = (void *)arg;
 461         clock_t deadline;
 462         int ret;
 463         bd_xfer_t xfer;

 465         deadline = ddi_get_lbolt() + drv_sectohz(3);
 465         deadline = ddi_get_lbolt() + (clock_t)drv_usectohz(3 * 1000000);
 466         (void) memset(&xfer, 0, sizeof (bd_xfer_t));
 467         xfer.x_nblks = 1;

 469         ret = ddi_dma_alloc_handle(sc->sc_dev, &vioblk_bd_dma_attr,
 470             DDI_DMA_SLEEP, NULL, &xfer.x_dmah);
 471         if (ret != DDI_SUCCESS)
 472                 goto out_alloc;

 474         ret = ddi_dma_addr_bind_handle(xfer.x_dmah, NULL, (caddr_t)&sc->devid,
 475             VIRTIO_BLK_ID_BYTES, DDI_DMA_READ | DDI_DMA_CONSISTENT,
 476             DDI_DMA_SLEEP, NULL, &xfer.x_dmac, &xfer.x_ndmac);
 477         if (ret != DDI_DMA_MAPPED) {
 478                 ret = DDI_FAILURE;
 479                 goto out_map;
 480         }

 482         mutex_enter(&sc->lock_devid);

 484         ret = vioblk_rw(sc, &xfer, VIRTIO_BLK_T_GET_ID,
 485             VIRTIO_BLK_ID_BYTES);
 486         if (ret) {
 487                 mutex_exit(&sc->lock_devid);
 488                 goto out_rw;
 489         }

 491         /* wait for reply */
 492         ret = cv_timedwait(&sc->cv_devid, &sc->lock_devid, deadline);
 493         mutex_exit(&sc->lock_devid);

 495         (void) ddi_dma_unbind_handle(xfer.x_dmah);
 496         ddi_dma_free_handle(&xfer.x_dmah);

 498         /* timeout */
 499         if (ret < 0) {
 500                 dev_err(devinfo, CE_WARN, "Cannot get devid from the device");
 501                 return (DDI_FAILURE);
 502         }

 504         ret = ddi_devid_init(devinfo, DEVID_ATA_SERIAL,
 505             VIRTIO_BLK_ID_BYTES, sc->devid, devid);
 506         if (ret != DDI_SUCCESS) {
 507                 dev_err(devinfo, CE_WARN, "Cannot build devid from the device");
 508                 return (ret);
 509         }

 511         dev_debug(sc->sc_dev, CE_NOTE,
 512             "devid %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x",
 513             sc->devid[0], sc->devid[1], sc->devid[2], sc->devid[3],
 514             sc->devid[4], sc->devid[5], sc->devid[6], sc->devid[7],
```



```
 515             sc->devid[8], sc->devid[9], sc->devid[10], sc->devid[11],
 516             sc->devid[12], sc->devid[13], sc->devid[14], sc->devid[15],
 517             sc->devid[16], sc->devid[17], sc->devid[18], sc->devid[19]);

 519         return (0);

 521 out_rw:
 522         (void) ddi_dma_unbind_handle(xfer.x_dmah);
 523 out_map:
 524         ddi_dma_free_handle(&xfer.x_dmah);
 525 out_alloc:
 526         return (ret);
 527 }
______unchanged_portion_omitted_
```

```
**********************************************************
   12688 Wed Aug 19 07:25:20 2015
new/usr/src/uts/common/io/xge/drv/xgell.h
XXXX introduce drv_sectohz
**********************************************************
```

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

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 *  Copyright (c) 2002-2005 Neterion, Inc.
 *  All right Reserved.
 *
 *  FileName :    xgell.h
 *
 *  Description:  Link Layer driver declaration
 *
 */

#ifndef _SYS_XGELL_H
#define _SYS_XGELL_H

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/param.h>
#include <sys/stropts.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/ksynch.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/debug.h>
#include <sys/pci.h>
#include <sys/ethernet.h>
#include <sys/vlan.h>
#include <sys/dlpi.h>
#include <sys/taskq.h>
#include <sys/cyclic.h>

#include <sys/pattr.h>
#include <sys/strsun.h>
```



```
#include <sys/mac_provider.h>
#include <sys/mac_ether.h>

#ifdef __cplusplus
extern "C" {
#endif

#define XGELL_DESC              "Xframe I/II 10Gb Ethernet"
#define XGELL_IFNAME            "xge"

#include <xgehal.h>

/*
 * The definition of XGELL_RX_BUFFER_RECYCLE_CACHE is an experimental value.
 * With this value, the lock contention between xgell_rx_buffer_recycle()
 * and xgell_rx_1b_compl() is reduced to great extent. And multiple rx rings
 * alleviate the lock contention further since each rx ring has its own mutex.
 */
#define XGELL_RX_BUFFER_RECYCLE_CACHE   XGE_HAL_RING_RXDS_PER_BLOCK(1) * 2
#define MSG_SIZE        64

/*
 * These default values can be overridden by vaules in xge.conf.
 * In xge.conf user has to specify actual (not percentages) values.
 */
#define XGELL_RX_BUFFER_TOTAL           XGE_HAL_RING_RXDS_PER_BLOCK(1) * 6
#define XGELL_RX_BUFFER_POST_HIWAT      XGE_HAL_RING_RXDS_PER_BLOCK(1) * 5

/*
 * Multiple rings configuration
 */
#define XGELL_RX_RING_MAIN                      0
#define XGELL_TX_RING_MAIN                      0

#define XGELL_RX_RING_NUM_MIN                   1
#define XGELL_TX_RING_NUM_MIN                   1
#define XGELL_RX_RING_NUM_MAX                   8
#define XGELL_TX_RING_NUM_MAX                   1 /* TODO */
#define XGELL_RX_RING_NUM_DEFAULT               XGELL_RX_RING_NUM_MAX
#define XGELL_TX_RING_NUM_DEFAULT               XGELL_TX_RING_NUM_MAX

#define XGELL_MINTR_NUM_MIN                     1
#define XGELL_MINTR_NUM_MAX                     \
        (XGELL_RX_RING_NUM_MAX + XGELL_TX_RING_NUM_MAX + 1)
#define XGELL_MINTR_NUM_DEFAULT                 XGELL_MINTR_NUM_MAX

#define XGELL_CONF_GROUP_POLICY_BASIC           0
#define XGELL_CONF_GROUP_POLICY_VIRT            1
#define XGELL_CONF_GROUP_POLICY_PERF            2
#if 0
#if defined(__sparc)
#define XGELL_CONF_GROUP_POLICY_DEFAULT         XGELL_CONF_GROUP_POLICY_PERF
#else
#define XGELL_CONF_GROUP_POLICY_DEFAULT         XGELL_CONF_GROUP_POLICY_VIRT
#endif
#else
/*
 * The _PERF configuration enable a fat group of all rx rings, as approachs
 * better fanout performance of the primary interface.
 */
#define XGELL_CONF_GROUP_POLICY_DEFAULT         XGELL_CONF_GROUP_POLICY_PERF
#endif

#define XGELL_TX_LEVEL_LOW      8
#define XGELL_TX_LEVEL_HIGH     32
```

```
128 #define XGELL_TX_LEVEL_CHECK    3
129 #define XGELL_MAX_RING_DEFAULT  8
130 #define XGELL_MAX_FIFO_DEFAULT  1

132 /* Control driver to copy or DMA inbound/outbound packets */
133 #if defined(__sparc)
134 #define XGELL_RX_DMA_LOWAT                      256
135 #define XGELL_TX_DMA_LOWAT                      512
136 #else
137 #define XGELL_RX_DMA_LOWAT                      256
138 #define XGELL_TX_DMA_LOWAT                      128
139 #endif

141 /*
142  * Try to collapse up to XGELL_RX_PKT_BURST packets into single mblk
143  * sequence before mac_rx() is called.
144  */
145 #define XGELL_RX_PKT_BURST                      32

147 /* About 1s */
148 #define XGE_DEV_POLL_TICKS                      drv_sectohz(1)
148 #define XGE_DEV_POLL_TICKS                      drv_usectohz(1000000)

150 #define XGELL_LSO_MAXLEN                        65535
151 #define XGELL_CONF_ENABLE_BY_DEFAULT            1
152 #define XGELL_CONF_DISABLE_BY_DEFAULT           0

154 /* LRO configuration */
155 #define XGE_HAL_DEFAULT_LRO_SG_SIZE             2 /* <=2 LRO fix not required */
156 #define XGE_HAL_DEFAULT_LRO_FRM_LEN             65535

158 /*
159  * Default values for tunables used in HAL. Please refer to xgehal-config.h
160  * for more details.
161  */
162 #define XGE_HAL_DEFAULT_USE_HARDCODE            -1

164 /* Bimodal adaptive schema defaults - ENABLED */
165 #define XGE_HAL_DEFAULT_BIMODAL_INTERRUPTS      -1
166 #define XGE_HAL_DEFAULT_BIMODAL_TIMER_LO_US     24
167 #define XGE_HAL_DEFAULT_BIMODAL_TIMER_HI_US     256

169 /* Interrupt moderation/utilization defaults */
170 #define XGE_HAL_DEFAULT_TX_URANGE_A             5
171 #define XGE_HAL_DEFAULT_TX_URANGE_B             15
172 #define XGE_HAL_DEFAULT_TX_URANGE_C             30
173 #define XGE_HAL_DEFAULT_TX_UFC_A                15
174 #define XGE_HAL_DEFAULT_TX_UFC_B                30
175 #define XGE_HAL_DEFAULT_TX_UFC_C                45
176 #define XGE_HAL_DEFAULT_TX_UFC_D                60
177 #define XGE_HAL_DEFAULT_TX_TIMER_CI_EN          1
178 #define XGE_HAL_DEFAULT_TX_TIMER_AC_EN          1
179 #define XGE_HAL_DEFAULT_TX_TIMER_VAL            10000
180 #define XGE_HAL_DEFAULT_INDICATE_MAX_PKTS_B     512 /* bimodal */
181 #define XGE_HAL_DEFAULT_INDICATE_MAX_PKTS_N     256 /* normal UFC */
182 #define XGE_HAL_DEFAULT_RX_URANGE_A             10
183 #define XGE_HAL_DEFAULT_RX_URANGE_B             30
184 #define XGE_HAL_DEFAULT_RX_URANGE_C             50
185 #define XGE_HAL_DEFAULT_RX_UFC_A                1
186 #define XGE_HAL_DEFAULT_RX_UFC_B_J              2
187 #define XGE_HAL_DEFAULT_RX_UFC_B_N              8
188 #define XGE_HAL_DEFAULT_RX_UFC_C_J              4
189 #define XGE_HAL_DEFAULT_RX_UFC_C_N              16
190 #define XGE_HAL_DEFAULT_RX_UFC_D                32
191 #define XGE_HAL_DEFAULT_RX_TIMER_AC_EN          1
192 #define XGE_HAL_DEFAULT_RX_TIMER_VAL            384
```



```
194 #define XGE_HAL_DEFAULT_FIFO_QUEUE_LENGTH_A     1024
195 #define XGE_HAL_DEFAULT_FIFO_QUEUE_LENGTH_J     2048
196 #define XGE_HAL_DEFAULT_FIFO_QUEUE_LENGTH_N     4096
197 #define XGE_HAL_DEFAULT_FIFO_QUEUE_INTR         0
198 #define XGE_HAL_DEFAULT_FIFO_RESERVE_THRESHOLD  0
199 #define XGE_HAL_DEFAULT_FIFO_MEMBLOCK_SIZE      PAGESIZE

201 /*
202  * This will force HAL to allocate extra copied buffer per TXDL which
203  * size calculated by formula:
204  *
205  *      (ALIGNMENT_SIZE * ALIGNED_FRAGS)
206  */
207 #define XGE_HAL_DEFAULT_FIFO_ALIGNMENT_SIZE     4096
208 #define XGE_HAL_DEFAULT_FIFO_MAX_ALIGNED_FRAGS  1
209 #if defined(__sparc)
210 #define XGE_HAL_DEFAULT_FIFO_FRAGS              64
211 #else
212 #define XGE_HAL_DEFAULT_FIFO_FRAGS              128
213 #endif
214 #define XGE_HAL_DEFAULT_FIFO_FRAGS_THRESHOLD    18

216 #define XGE_HAL_DEFAULT_RING_QUEUE_BLOCKS       2
217 #define XGE_HAL_RING_QUEUE_BUFFER_MODE_DEFAULT  1
218 #define XGE_HAL_DEFAULT_BACKOFF_INTERVAL_US     64
219 #define XGE_HAL_DEFAULT_RING_PRIORITY           0
220 #define XGE_HAL_DEFAULT_RING_MEMBLOCK_SIZE      PAGESIZE

222 #define XGE_HAL_DEFAULT_RING_NUM                8
223 #define XGE_HAL_DEFAULT_TMAC_UTIL_PERIOD        5
224 #define XGE_HAL_DEFAULT_RMAC_UTIL_PERIOD        5
225 #define XGE_HAL_DEFAULT_RMAC_HIGH_PTIME         65535
226 #define XGE_HAL_DEFAULT_MC_PAUSE_THRESHOLD_Q0Q3 187
227 #define XGE_HAL_DEFAULT_MC_PAUSE_THRESHOLD_Q4Q7 187
228 #define XGE_HAL_DEFAULT_RMAC_PAUSE_GEN_EN       1
229 #define XGE_HAL_DEFAULT_RMAC_PAUSE_GEN_DIS      0
230 #define XGE_HAL_DEFAULT_RMAC_PAUSE_RCV_EN       1
231 #define XGE_HAL_DEFAULT_RMAC_PAUSE_RCV_DIS      0
232 #define XGE_HAL_DEFAULT_INITIAL_MTU             XGE_HAL_DEFAULT_MTU /* 1500 */
233 #define XGE_HAL_DEFAULT_ISR_POLLING_CNT         0
234 #define XGE_HAL_DEFAULT_LATENCY_TIMER           255
235 #define XGE_HAL_DEFAULT_SHARED_SPLITS           0
236 #define XGE_HAL_DEFAULT_STATS_REFRESH_TIME      1

238 #if defined(__sparc)
239 #define XGE_HAL_DEFAULT_MMRB_COUNT              XGE_HAL_MAX_MMRB_COUNT
240 #define XGE_HAL_DEFAULT_SPLIT_TRANSACTION       XGE_HAL_EIGHT_SPLIT_TRANSACTION
241 #else
242 #define XGE_HAL_DEFAULT_MMRB_COUNT              1 /* 1k */
243 #define XGE_HAL_DEFAULT_SPLIT_TRANSACTION       XGE_HAL_TWO_SPLIT_TRANSACTION
244 #endif

246 /*
247  * Default the size of buffers allocated for ndd interface functions
248  */
249 #define XGELL_STATS_BUFSIZE                     8192
250 #define XGELL_PCICONF_BUFSIZE                   2048
251 #define XGELL_ABOUT_BUFSIZE                     512
252 #define XGELL_IOCTL_BUFSIZE                     64
253 #define XGELL_DEVCONF_BUFSIZE                   8192

255 /*
256  * Multiple mac address definitions
257  *
258  * We'll use whole MAC Addresses Configuration Memory for unicast addresses,
```

```
 259  * since current multicast implementation in HAL is by enabling promise mode.
 260  */
 261 #define XGE_RX_MULTI_MAC_ADDRESSES_MAX          8 /* per ring group */

 263 typedef struct {
 264         int rx_pkt_burst;
 265         int rx_buffer_total;
 266         int rx_buffer_post_hiwat;
 267         int rx_dma_lowat;
 268         int tx_dma_lowat;
 269         int lso_enable;
 270         int msix_enable;
 271         int grouping;
 272 } xgell_config_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   32246 Wed Aug 19 07:25:20 2015
new/usr/src/uts/common/os/ddi.c
XXXX introduce drv_sectohz
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */
#endif /* ! codereview */

/*
 * UNIX Device Driver Interface functions
 *
 * This file contains functions that are to be added to the kernel
 * to put the interface presented to drivers in conformance with
 * the DDI standard. Of the functions added to the kernel, 17 are
 * function equivalents of existing macros in sysmacros.h,
 * stream.h, and param.h
 *
 * 17 additional functions -- drv_getparm(), drv_setparm(),
 * getrbuf(), freerbuf(),
 * getemajor(), geteminor(), etoimajor(), itoemajor(), drv_usectohz(),
 * drv_hztousec(), drv_usecwait(), drv_priv(), and kvtoppid() --
 * are specified by DDI to exist in the kernel and are implemented here.
 *
 * Note that putnext() and put() are not in this file. The C version of
 * these routines are in uts/common/os/putnext.c and assembly versions
 * might exist for some architectures.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/time.h>
#include <sys/systm.h>
#include <sys/cpuvar.h>
#include <sys/signal.h>
#include <sys/pcb.h>
```



```
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/buf.h>
#include <sys/proc.h>
#include <sys/cmn_err.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/uio.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/poll.h>
#include <sys/session.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/esunddi.h>
#include <sys/mkdev.h>
#include <sys/debug.h>
#include <sys/vtrace.h>

/*
 * return internal major number corresponding to device
 * number (new format) argument
 */
major_t
getmajor(dev_t dev)
{
#ifdef _LP64
        return ((major_t)((dev >> NBITSMINOR64) & MAXMAJ64));
#else
        return ((major_t)((dev >> NBITSMINOR) & MAXMAJ));
#endif
}

/*
 * return external major number corresponding to device
 * number (new format) argument
 */
major_t
getemajor(dev_t dev)
{
#ifdef _LP64
        return ((major_t)((dev >> NBITSMINOR64) & MAXMAJ64));
#else
        return ((major_t)((dev >> NBITSMINOR) & MAXMAJ));
#endif
}

/*
 * return internal minor number corresponding to device
 * number (new format) argument
 */
minor_t
getminor(dev_t dev)
{
#ifdef _LP64
        return ((minor_t)(dev & MAXMIN64));
#else
        return ((minor_t)(dev & MAXMIN));
#endif
}

/*
 * return external minor number corresponding to device
```

```
 * number (new format) argument
 */
minor_t
geteminor(dev_t dev)
{
#ifdef _LP64
        return ((minor_t)(dev & MAXMIN64));
#else
        return ((minor_t)(dev & MAXMIN));
#endif
}

/*
 * return internal major number corresponding to external
 * major number.
 */
int
etoimajor(major_t emajnum)
{
#ifdef _LP64
        if (emajnum >= devcnt)
                return (-1); /* invalid external major */
#else
        if (emajnum > MAXMAJ || emajnum >= devcnt)
                return (-1); /* invalid external major */
#endif
        return ((int)emajnum);
}

/*
 * return external major number corresponding to internal
 * major number argument or -1 if no external major number
 * can be found after lastemaj that maps to the internal
 * major number. Pass a lastemaj val of -1 to start
 * the search initially. (Typical use of this function is
 * of the form:
 *
 *      lastemaj = -1;
 *      while ((lastemaj = itoemajor(imag, lastemaj)) != -1)
 *              { process major number }
 */
int
itoemajor(major_t imajnum, int lastemaj)
{
        if (imajnum >= devcnt)
                return (-1);

        /*
         * if lastemaj == -1 then start from beginning of
         * the (imaginary) MAJOR table
         */
        if (lastemaj < -1)
                return (-1);

        /*
         * given that there's a 1-1 mapping of internal to external
         * major numbers, searching is somewhat pointless ... let's
         * just go there directly.
         */
        if (++lastemaj < devcnt && imajnum < devcnt)
                return (imajnum);
        return (-1);
}

/*
 * encode external major and minor number arguments into a
```



```
 * new format device number
 */
dev_t
makedevice(major_t maj, minor_t minor)
{
#ifdef _LP64
        return (((dev_t)maj << NBITSMINOR64) | (minor & MAXMIN64));
#else
        return (((dev_t)maj << NBITSMINOR) | (minor & MAXMIN));
#endif
}

/*
 * cmpdev - compress new device format to old device format
 */
o_dev_t
cmpdev(dev_t dev)
{
        major_t major_d;
        minor_t minor_d;

#ifdef _LP64
        major_d = dev >> NBITSMINOR64;
        minor_d = dev & MAXMIN64;
#else
        major_d = dev >> NBITSMINOR;
        minor_d = dev & MAXMIN;
#endif
        if (major_d > OMAXMAJ || minor_d > OMAXMIN)
                return ((o_dev_t)NODEV);
        return ((o_dev_t)((major_d << ONBITSMINOR) | minor_d));
}

dev_t
expdev(dev_t dev)
{
        major_t major_d;
        minor_t minor_d;

        major_d = ((dev >> ONBITSMINOR) & OMAXMAJ);
        minor_d = (dev & OMAXMIN);
#ifdef _LP64
        return ((((dev_t)major_d << NBITSMINOR64) | minor_d));
#else
        return ((((dev_t)major_d << NBITSMINOR) | minor_d));
#endif
}

/*
 * return true (1) if the message type input is a data
 * message type, 0 otherwise
 */
#undef datamsg
int
datamsg(unsigned char db_type)
{
        return (db_type == M_DATA || db_type == M_PROTO ||
            db_type == M_PCPROTO || db_type == M_DELAY);
}

/*
 * return a pointer to the other queue in the queue pair of qp
 */
queue_t *
OTHERQ(queue_t *q)
{
```

```
260         return (_OTHERQ(q));
261 }

263 /*
264  * return a pointer to the read queue in the queue pair of qp.
265  */
266 queue_t *
267 RD(queue_t *q)
268 {
269                 return (_RD(q));

271 }

273 /*
274  * return a pointer to the write queue in the queue pair of qp.
275  */
276 int
277 SAMESTR(queue_t *q)
278 {
279         return (_SAMESTR(q));
280 }

282 /*
283  * return a pointer to the write queue in the queue pair of qp.
284  */
285 queue_t *
286 WR(queue_t *q)
287 {
288         return (_WR(q));
289 }

291 /*
292  * store value of kernel parameter associated with parm
293  */
294 int
295 drv_getparm(unsigned int parm, void *valuep)
296 {
297         proc_t  *p = curproc;
298         time_t  now;

300         switch (parm) {
301         case UPROCP:
302                 *(proc_t **)valuep = p;
303                 break;
304         case PPGRP:
305                 mutex_enter(&p->p_lock);
306                 *(pid_t *)valuep = p->p_pgrp;
307                 mutex_exit(&p->p_lock);
308                 break;
309         case LBOLT:
310                 *(clock_t *)valuep = ddi_get_lbolt();
311                 break;
312         case TIME:
313                 if ((now = gethrestime_sec()) == 0) {
314                         timestruc_t ts;
315                         mutex_enter(&tod_lock);
316                         ts = tod_get();
317                         mutex_exit(&tod_lock);
318                         *(time_t *)valuep = ts.tv_sec;
319                 } else {
320                         *(time_t *)valuep = now;
321                 }
322                 break;
323         case PPID:
324                 *(pid_t *)valuep = p->p_pid;
325                 break;
```



```
326         case PSID:
327                 mutex_enter(&p->p_splock);
328                 *(pid_t *)valuep = p->p_sessp->s_sid;
329                 mutex_exit(&p->p_splock);
330                 break;
331         case UCRED:
332                 *(cred_t **)valuep = CRED();
333                 break;
334         default:
335                 return (-1);
336         }

338         return (0);
339 }

341 /*
342  * set value of kernel parameter associated with parm
343  */
344 int
345 drv_setparm(unsigned int parm, unsigned long value)
346 {
347         switch (parm) {
348         case SYSRINT:
349                 CPU_STATS_ADDQ(CPU, sys, rcvint, value);
350                 break;
351         case SYSXINT:
352                 CPU_STATS_ADDQ(CPU, sys, xmtint, value);
353                 break;
354         case SYSMINT:
355                 CPU_STATS_ADDQ(CPU, sys, mdmint, value);
356                 break;
357         case SYSRAWC:
358                 CPU_STATS_ADDQ(CPU, sys, rawch, value);
359                 break;
360         case SYSCANC:
361                 CPU_STATS_ADDQ(CPU, sys, canch, value);
362                 break;
363         case SYSOUTC:
364                 CPU_STATS_ADDQ(CPU, sys, outch, value);
365                 break;
366         default:
367                 return (-1);
368         }

370         return (0);
371 }

373 /*
374  * allocate space for buffer header and return pointer to it.
375  * preferred means of obtaining space for a local buf header.
376  * returns pointer to buf upon success, NULL for failure
377  */
378 struct buf *
379 getrbuf(int sleep)
380 {
381         struct buf *bp;

383         bp = kmem_alloc(sizeof (struct buf), sleep);
384         if (bp == NULL)
385                 return (NULL);
386         bioinit(bp);

388         return (bp);
389 }

391 /*
```

```
 * free up space allocated by getrbuf()
 */
void
freerbuf(struct buf *bp)
{
        biofini(bp);
        kmem_free(bp, sizeof (struct buf));
}

/*
 * convert byte count input to logical page units
 * (byte counts that are not a page-size multiple
 * are rounded down)
 */
pgcnt_t
btop(size_t numbytes)
{
        return (numbytes >> PAGESHIFT);
}

/*
 * convert byte count input to logical page units
 * (byte counts that are not a page-size multiple
 * are rounded up)
 */
pgcnt_t
btopr(size_t numbytes)
{
        return ((numbytes + PAGEOFFSET) >> PAGESHIFT);
}

/*
 * convert size in pages to bytes.
 */
size_t
ptob(pgcnt_t numpages)
{
        return (numpages << PAGESHIFT);
}

#define MAXCLOCK_T LONG_MAX

/*
 * Convert from system time units (hz) to microseconds.
 *
 * If ticks <= 0, return 0.
 * If converting ticks to usecs would overflow, return MAXCLOCK_T.
 * Otherwise, convert ticks to microseconds.
 */
clock_t
drv_hztousec(clock_t ticks)
{
        if (ticks <= 0)
                return (0);

        if (ticks > MAXCLOCK_T / usec_per_tick)
                return (MAXCLOCK_T);

        return (TICK_TO_USEC(ticks));
}


/*
 * Convert from microseconds to system time units (hz), rounded up.
 *
 * If ticks <= 0, return 0.
```



```
 * Otherwise, convert microseconds to ticks, rounding up.
 */
clock_t
drv_usectohz(clock_t microsecs)
{
        if (microsecs <= 0)
                return (0);

        return (USEC_TO_TICK_ROUNDUP(microsecs));
}

/*
 * Convert from seconds to system time units (hz).
 *
 * If secs <= 0, return 0.
 * Otherwise, convert seconds to ticks, rounding up.
 */
clock_t
drv_sectohz(clock_t secs)
{
        if (secs <= 0)
                return (0);

        return (SEC_TO_TICK(secs));
#endif /* ! codereview */
}

#ifdef  sun
/*
 * drv_usecwait implemented in each architecture's machine
 * specific code somewhere. For sparc, it is the alternate entry
 * to usec_delay (eventually usec_delay goes away). See
 * sparc/os/ml/sparc_subr.s
 */
#endif

/*
 * bcanputnext, canputnext assume called from timeout, bufcall,
 * or esballoc free routines.  since these are driven by
 * clock interrupts, instead of system calls the appropriate plumbing
 * locks have not been acquired.
 */
int
bcanputnext(queue_t *q, unsigned char band)
{
        int     ret;

        claimstr(q);
        ret = bcanput(q->q_next, band);
        releasestr(q);
        return (ret);
}

int
canputnext(queue_t *q)
{
        queue_t *qofsq = q;
        struct stdata *stp = STREAM(q);
        kmutex_t *sdlock;

        TRACE_1(TR_FAC_STREAMS_FR, TR_CANPUTNEXT_IN,
            "canputnext?:%p\n", q);

        if (stp->sd_ciputctrl != NULL) {
                int ix = CPU->cpu_seqid & stp->sd_nciputctrl;
                sdlock = &stp->sd_ciputctrl[ix].ciputctrl_lock;
```

```
524                 mutex_enter(sdlock);
525         } else
526                 mutex_enter(sdlock = &stp->sd_reflock);
527
528         /* get next module forward with a service queue */
529         q = q->q_next->q_nfsrv;
530         ASSERT(q != NULL);
531
532         /* this is for loopback transports, they should not do a canputnext */
533         ASSERT(STRMATED(q->q_stream) || STREAM(q) == STREAM(qofsq));
534
535         if (!(q->q_flag & QFULL)) {
536                 mutex_exit(sdlock);
537                 TRACE_2(TR_FAC_STREAMS_FR, TR_CANPUTNEXT_OUT,
538                     "canputnext:%p %d", q, 1);
539                 return (1);
540         }
541
542         if (sdlock != &stp->sd_reflock) {
543                 mutex_exit(sdlock);
544                 mutex_enter(&stp->sd_reflock);
545         }
546
547         /* the above is the most frequently used path */
548         stp->sd_refcnt++;
549         ASSERT(stp->sd_refcnt != 0);    /* Wraparound */
550         mutex_exit(&stp->sd_reflock);
551
552         mutex_enter(QLOCK(q));
553         if (q->q_flag & QFULL) {
554                 q->q_flag |= QWANTW;
555                 mutex_exit(QLOCK(q));
556                 TRACE_2(TR_FAC_STREAMS_FR, TR_CANPUTNEXT_OUT,
557                     "canputnext:%p %d", q, 0);
558                 releasestr(qofsq);
559
560                 return (0);
561         }
562         mutex_exit(QLOCK(q));
563         TRACE_2(TR_FAC_STREAMS_FR, TR_CANPUTNEXT_OUT, "canputnext:%p %d", q, 1);
564         releasestr(qofsq);
565
566         return (1);
567 }
568
569
570 /*
571  * Open has progressed to the point where it is safe to send/receive messages.
572  *
573  * "qprocson enables the put and service routines of the driver
574  * or module... Prior to the call to qprocson, the put and service
575  * routines of a newly pushed module or newly opened driver are
576  * disabled.  For the module, messages flow around it as if it
577  * were not present in the stream... qprocson must be called by
578  * the first open of a module or driver after allocation and
579  * initialization of any resource on which the put and service
580  * routines depend."
581  *
582  * Note that before calling qprocson a module/driver could itself cause its
583  * put or service procedures to be run by using put() or qenable().
584  */
585 void
586 qprocson(queue_t *q)
587 {
588         ASSERT(q->q_flag & QREADR);
589         /*
```



```
590          * Do not call insertq() if it is a re-open.  But if _QINSERTING
591          * is set, q_next will not be NULL and we need to call insertq().
592          */
593         if ((q->q_next == NULL && WR(q)->q_next == NULL) ||
594             (q->q_flag & _QINSERTING))
595                 insertq(STREAM(q), q);
596 }
597
598 /*
599  * Close has reached a point where it can no longer allow put/service
600  * into the queue.
601  *
602  * "qprocsoff disables the put and service routines of the driver
603  * or module... When the routines are disabled in a module, messages
604  * flow around the module as if it were not present in the stream.
605  * qprocsoff must be called by the close routine of a driver or module
606  * before deallocating any resources on which the driver/module's
607  * put and service routines depend.  qprocsoff will remove the
608  * queue's service routines from the list of service routines to be
609  * run and waits until any concurrent put or service routines are
610  * finished."
611  *
612  * Note that after calling qprocsoff a module/driver could itself cause its
613  * put procedures to be run by using put().
614  */
615 void
616 qprocsoff(queue_t *q)
617 {
618         ASSERT(q->q_flag & QREADR);
619         if (q->q_flag & QWCLOSE) {
620                 /* Called more than once */
621                 return;
622         }
623         disable_svc(q);
624         removeq(q);
625 }
626
627 /*
628  * "freezestr() freezes the state of the entire STREAM  containing
629  *  the  queue  pair  q.  A frozen STREAM blocks any thread
630  *  attempting to enter any open, close, put or service  routine
631  *  belonging  to  any  queue instance in the STREAM, and blocks
632  *  any thread currently within the STREAM if it attempts to put
633  *  messages  onto  or take messages off of any queue within the
634  *  STREAM (with the sole exception  of  the  caller).   Threads
635  *  blocked  by  this  mechanism  remain  so until the STREAM is
636  *  thawed by a call to unfreezestr().
637  *
638  * Use strblock to set SQ_FROZEN in all syncqs in the stream (prevents
639  * further entry into put, service, open, and close procedures) and
640  * grab (and hold) all the QLOCKs in the stream (to block putq, getq etc.)
641  *
642  * Note: this has to be the only code that acquires one QLOCK while holding
643  * another QLOCK (otherwise we would have locking hirarchy/ordering violations.)
644  */
645 void
646 freezestr(queue_t *q)
647 {
648         struct stdata *stp = STREAM(q);
649
650         /*
651          * Increment refcnt to prevent q_next from changing during the strblock
652          * as well as while the stream is frozen.
653          */
654         claimstr(RD(q));
```

```
656         strblock(q);
657         ASSERT(stp->sd_freezer == NULL);
658         stp->sd_freezer = curthread;
659         for (q = stp->sd_wrq; q != NULL; q = SAMESTR(q) ? q->q_next : NULL) {
660                 mutex_enter(QLOCK(q));
661                 mutex_enter(QLOCK(RD(q)));
662         }
663 }

665 /*
666  * Undo what freezestr did.
667  * Have to drop the QLOCKs before the strunblock since strunblock will
668  * potentially call other put procedures.
669  */
670 void
671 unfreezestr(queue_t *q)
672 {
673         struct stdata *stp = STREAM(q);
674         queue_t *q1;

676         for (q1 = stp->sd_wrq; q1 != NULL;
677             q1 = SAMESTR(q1) ? q1->q_next : NULL) {
678                 mutex_exit(QLOCK(q1));
679                 mutex_exit(QLOCK(RD(q1)));
680         }
681         ASSERT(stp->sd_freezer == curthread);
682         stp->sd_freezer = NULL;
683         strunblock(q);
684         releasestr(RD(q));
685 }

687 /*
688  * Used by open and close procedures to "sleep" waiting for messages to
689  * arrive. Note: can only be used in open and close procedures.
690  *
691  * Lower the gate and let in either messages on the syncq (if there are
692  * any) or put/service procedures.
693  *
694  * If the queue has an outer perimeter this will not prevent entry into this
695  * syncq (since outer_enter does not set SQ_WRITER on the syncq that gets the
696  * exclusive access to the outer perimeter.)
697  *
698  * Return 0 is the cv_wait_sig was interrupted; otherwise 1.
699  *
700  * It only makes sense to grab sq_putlocks for !SQ_CIOC sync queues because
701  * otherwise put entry points were not blocked in the first place. if this is
702  * SQ_CIOC then qwait is used to wait for service procedure to run since syncq
703  * is always SQ_CIPUT if it is SQ_CIOC.
704  *
705  * Note that SQ_EXCL is dropped and SQ_WANTEXITWAKEUP set in sq_flags
706  * atomically under sq_putlocks to make sure putnext will not miss a pending
707  * wakeup.
708  */
709 int
710 qwait_sig(queue_t *q)
711 {
712         syncq_t         *sq, *outer;
713         uint_t          flags;
714         int             ret = 1;
715         int             is_sq_cioc;

717         /*
718          * Perform the same operations as a leavesq(sq, SQ_OPENCLOSE)
719          * while detecting all cases where the perimeter is entered
720          * so that qwait_sig can return to the caller.
721          *
```



```
722          * Drain the syncq if possible. Otherwise reset SQ_EXCL and
723          * wait for a thread to leave the syncq.
724          */
725         sq = q->q_syncq;
726         ASSERT(sq);
727         is_sq_cioc = (sq->sq_type & SQ_CIOC) ? 1 : 0;
728         ASSERT(sq->sq_outer == NULL || sq->sq_outer->sq_flags & SQ_WRITER);
729         outer = sq->sq_outer;
730         /*
731          * XXX this does not work if there is only an outer perimeter.
732          * The semantics of qwait/qwait_sig are undefined in this case.
733          */
734         if (outer)
735                 outer_exit(outer);

737         mutex_enter(SQLOCK(sq));
738         if (is_sq_cioc == 0) {
739                 SQ_PUTLOCKS_ENTER(sq);
740         }
741         flags = sq->sq_flags;
742         /*
743          * Drop SQ_EXCL and sq_count but hold the SQLOCK
744          * to prevent any undetected entry and exit into the perimeter.
745          */
746         ASSERT(sq->sq_count > 0);
747         sq->sq_count--;

749         if (is_sq_cioc == 0) {
750                 ASSERT(flags & SQ_EXCL);
751                 flags &= ~SQ_EXCL;
752         }
753         /*
754          * Unblock any thread blocked in an entersq or outer_enter.
755          * Note: we do not unblock a thread waiting in qwait/qwait_sig,
756          * since that could lead to livelock with two threads in
757          * qwait for the same (per module) inner perimeter.
758          */
759         if (flags & SQ_WANTWAKEUP) {
760                 cv_broadcast(&sq->sq_wait);
761                 flags &= ~SQ_WANTWAKEUP;
762         }
763         sq->sq_flags = flags;
764         if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
765                 if (is_sq_cioc == 0) {
766                         SQ_PUTLOCKS_EXIT(sq);
767                 }
768                 /* drain_syncq() drops SQLOCK */
769                 drain_syncq(sq);
770                 ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
771                 entersq(sq, SQ_OPENCLOSE);
772                 return (1);
773         }
774         /*
775          * Sleep on sq_exitwait to only be woken up when threads leave the
776          * put or service procedures. We can not sleep on sq_wait since an
777          * outer_exit in a qwait running in the same outer perimeter would
778          * cause a livelock "ping-pong" between two or more qwait'ers.
779          */
780         do {
781                 sq->sq_flags |= SQ_WANTEXWAKEUP;
782                 if (is_sq_cioc == 0) {
783                         SQ_PUTLOCKS_EXIT(sq);
784                 }
785                 ret = cv_wait_sig(&sq->sq_exitwait, SQLOCK(sq));
786                 if (is_sq_cioc == 0) {
787                         SQ_PUTLOCKS_ENTER(sq);
```

```
788                 }
789         } while (ret && (sq->sq_flags & SQ_WANTEXWAKEUP));
790         if (is_sq_cioc == 0) {
791                 SQ_PUTLOCKS_EXIT(sq);
792         }
793         mutex_exit(SQLOCK(sq));

795         /*
796          * Re-enter the perimeters again
797          */
798         entersq(sq, SQ_OPENCLOSE);
799         return (ret);
800 }

802 /*
803  * Used by open and close procedures to "sleep" waiting for messages to
804  * arrive. Note: can only be used in open and close procedures.
805  *
806  * Lower the gate and let in either messages on the syncq (if there are
807  * any) or put/service procedures.
808  *
809  * If the queue has an outer perimeter this will not prevent entry into this
810  * syncq (since outer_enter does not set SQ_WRITER on the syncq that gets the
811  * exclusive access to the outer perimeter.)
812  *
813  * It only makes sense to grab sq_putlocks for !SQ_CIOC sync queues because
814  * otherwise put entry points were not blocked in the first place. if this is
815  * SQ_CIOC then qwait is used to wait for service procedure to run since syncq
816  * is always SQ_CIPUT if it is SQ_CIOC.
817  *
818  * Note that SQ_EXCL is dropped and SQ_WANTEXITWAKEUP set in sq_flags
819  * atomically under sq_putlocks to make sure putnext will not miss a pending
820  * wakeup.
821  */
822 void
823 qwait(queue_t *q)
824 {
825         syncq_t         *sq, *outer;
826         uint_t          flags;
827         int             is_sq_cioc;

829         /*
830          * Perform the same operations as a leavesq(sq, SQ_OPENCLOSE)
831          * while detecting all cases where the perimeter is entered
832          * so that qwait can return to the caller.
833          *
834          * Drain the syncq if possible. Otherwise reset SQ_EXCL and
835          * wait for a thread to leave the syncq.
836          */
837         sq = q->q_syncq;
838         ASSERT(sq);
839         is_sq_cioc = (sq->sq_type & SQ_CIOC) ? 1 : 0;
840         ASSERT(sq->sq_outer == NULL || sq->sq_outer->sq_flags & SQ_WRITER);
841         outer = sq->sq_outer;
842         /*
843          * XXX this does not work if there is only an outer perimeter.
844          * The semantics of qwait/qwait_sig are undefined in this case.
845          */
846         if (outer)
847                 outer_exit(outer);

849         mutex_enter(SQLOCK(sq));
850         if (is_sq_cioc == 0) {
851                 SQ_PUTLOCKS_ENTER(sq);
852         }
853         flags = sq->sq_flags;
```



```
854         /*
855          * Drop SQ_EXCL and sq_count but hold the SQLOCK
856          * to prevent any undetected entry and exit into the perimeter.
857          */
858         ASSERT(sq->sq_count > 0);
859         sq->sq_count--;

861         if (is_sq_cioc == 0) {
862                 ASSERT(flags & SQ_EXCL);
863                 flags &= ~SQ_EXCL;
864         }
865         /*
866          * Unblock any thread blocked in an entersq or outer_enter.
867          * Note: we do not unblock a thread waiting in qwait/qwait_sig,
868          * since that could lead to livelock with two threads in
869          * qwait for the same (per module) inner perimeter.
870          */
871         if (flags & SQ_WANTWAKEUP) {
872                 cv_broadcast(&sq->sq_wait);
873                 flags &= ~SQ_WANTWAKEUP;
874         }
875         sq->sq_flags = flags;
876         if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
877                 if (is_sq_cioc == 0) {
878                         SQ_PUTLOCKS_EXIT(sq);
879                 }
880                 /* drain_syncq() drops SQLOCK */
881                 drain_syncq(sq);
882                 ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
883                 entersq(sq, SQ_OPENCLOSE);
884                 return;
885         }
886         /*
887          * Sleep on sq_exitwait to only be woken up when threads leave the
888          * put or service procedures. We can not sleep on sq_wait since an
889          * outer_exit in a qwait running in the same outer perimeter would
890          * cause a livelock "ping-pong" between two or more qwait'ers.
891          */
892         do {
893                 sq->sq_flags |= SQ_WANTEXWAKEUP;
894                 if (is_sq_cioc == 0) {
895                         SQ_PUTLOCKS_EXIT(sq);
896                 }
897                 cv_wait(&sq->sq_exitwait, SQLOCK(sq));
898                 if (is_sq_cioc == 0) {
899                         SQ_PUTLOCKS_ENTER(sq);
900                 }
901         } while (sq->sq_flags & SQ_WANTEXWAKEUP);
902         if (is_sq_cioc == 0) {
903                 SQ_PUTLOCKS_EXIT(sq);
904         }
905         mutex_exit(SQLOCK(sq));

907         /*
908          * Re-enter the perimeters again
909          */
910         entersq(sq, SQ_OPENCLOSE);
911 }

913 /*
914  * Used for the synchronous streams entrypoints when sleeping outside
915  * the perimeters. Must never be called from regular put entrypoint.
916  *
917  * There's no need to grab sq_putlocks here (which only exist for CIPUT sync
918  * queues). If it is CIPUT sync queue put entry points were not blocked in the
919  * first place by rwnext/infonext which are treated as put entrypoints for
```

```
 * permiter syncronization purposes.
 *
 * Consolidation private.
 */
boolean_t
qwait_rw(queue_t *q)
{
        syncq_t         *sq;
        ulong_t         flags;
        boolean_t       gotsignal = B_FALSE;

        /*
         * Perform the same operations as a leavesq(sq, SQ_PUT)
         * while detecting all cases where the perimeter is entered
         * so that qwait_rw can return to the caller.
         *
         * Drain the syncq if possible. Otherwise reset SQ_EXCL and
         * wait for a thread to leave the syncq.
         */
        sq = q->q_syncq;
        ASSERT(sq);

        mutex_enter(SQLOCK(sq));
        flags = sq->sq_flags;
        /*
         * Drop SQ_EXCL and sq_count but hold the SQLOCK until to prevent any
         * undetected entry and exit into the perimeter.
         */
        ASSERT(sq->sq_count > 0);
        sq->sq_count--;
        if (!(sq->sq_type & SQ_CIPUT)) {
                ASSERT(flags & SQ_EXCL);
                flags &= ~SQ_EXCL;
        }
        /*
         * Unblock any thread blocked in an entersq or outer_enter.
         * Note: we do not unblock a thread waiting in qwait/qwait_sig,
         * since that could lead to livelock with two threads in
         * qwait for the same (per module) inner perimeter.
         */
        if (flags & SQ_WANTWAKEUP) {
                cv_broadcast(&sq->sq_wait);
                flags &= ~SQ_WANTWAKEUP;
        }
        sq->sq_flags = flags;
        if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
                /* drain_syncq() drops SQLOCK */
                drain_syncq(sq);
                ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
                entersq(sq, SQ_PUT);
                return (B_FALSE);
        }
        /*
         * Sleep on sq_exitwait to only be woken up when threads leave the
         * put or service procedures. We can not sleep on sq_wait since an
         * outer_exit in a qwait running in the same outer perimeter would
         * cause a livelock "ping-pong" between two or more qwait'ers.
         */
        do {
                sq->sq_flags |= SQ_WANTEXWAKEUP;
                if (cv_wait_sig(&sq->sq_exitwait, SQLOCK(sq)) <= 0) {
                        sq->sq_flags &= ~SQ_WANTEXWAKEUP;
                        gotsignal = B_TRUE;
                        break;
                }
        } while (sq->sq_flags & SQ_WANTEXWAKEUP);
```



```
        mutex_exit(SQLOCK(sq));

        /*
         * Re-enter the perimeters again
         */
        entersq(sq, SQ_PUT);
        return (gotsignal);
}

/*
 * Asynchronously upgrade to exclusive access at either the inner or
 * outer perimeter.
 */
void
qwriter(queue_t *q, mblk_t *mp, void (*func)(), int perim)
{
        if (perim == PERIM_INNER)
                qwriter_inner(q, mp, func);
        else if (perim == PERIM_OUTER)
                qwriter_outer(q, mp, func);
        else
                panic("qwriter: wrong \"perimeter\" parameter");
}

/*
 * Schedule a synchronous streams timeout
 */
timeout_id_t
qtimeout(queue_t *q, void (*func)(void *), void *arg, clock_t tim)
{
        syncq_t         *sq;
        callbparams_t   *cbp;
        timeout_id_t    tid;

        sq = q->q_syncq;
        /*
         * you don't want the timeout firing before its params are set up
         * callbparams_alloc() acquires SQLOCK(sq)
         * qtimeout() can't fail and can't sleep, so panic if memory is not
         * available.
         */
        cbp = callbparams_alloc(sq, func, arg, KM_NOSLEEP | KM_PANIC);
        /*
         * the callbflags in the sq use the same flags. They get anded
         * in the callbwrapper to determine if a qun* of this callback type
         * is required. This is not a request to cancel.
         */
        cbp->cbp_flags = SQ_CANCEL_TOUT;
        /* check new timeout version return codes */
        tid = timeout(qcallbwrapper, cbp, tim);
        cbp->cbp_id = (callbparams_id_t)tid;
        mutex_exit(SQLOCK(sq));
        /* use local id because the cbp memory could be free by now */
        return (tid);
}

bufcall_id_t
qbufcall(queue_t *q, size_t size, uint_t pri, void (*func)(void *), void *arg)
{
        syncq_t         *sq;
        callbparams_t   *cbp;
        bufcall_id_t    bid;

        sq = q->q_syncq;
        /*
         * you don't want the timeout firing before its params are set up
```

```
1052          * callbparams_alloc() acquires SQLOCK(sq) if successful.
1053          */
1054         cbp = callbparams_alloc(sq, func, arg, KM_NOSLEEP);
1055         if (cbp == NULL)
1056                 return ((bufcall_id_t)0);

1058         /*
1059          * the callbflags in the sq use the same flags. They get anded
1060          * in the callbwrapper to determine if a qun* of this callback type
1061          * is required. This is not a request to cancel.
1062          */
1063         cbp->cbp_flags = SQ_CANCEL_BUFCALL;
1064         /* check new timeout version return codes */
1065         bid = bufcall(size, pri, qcallbwrapper, cbp);
1066         cbp->cbp_id = (callbparams_id_t)bid;
1067         if (bid == 0) {
1068                 callbparams_free(sq, cbp);
1069         }
1070         mutex_exit(SQLOCK(sq));
1071         /* use local id because the params memory could be free by now */
1072         return (bid);
1073 }

1075 /*
1076  * cancel a timeout callback which enters the inner perimeter.
1077  * cancelling of all callback types on a given syncq is serialized.
1078  * the SQ_CALLB_BYPASSED flag indicates that the callback fn did
1079  * not execute. The quntimeout return value needs to reflect this.
1080  * As with out existing callback programming model - callbacks must
1081  * be cancelled before a close completes - so ensuring that the sq
1082  * is valid when the callback wrapper is executed.
1083  */
1084 clock_t
1085 quntimeout(queue_t *q, timeout_id_t id)
1086 {
1087         syncq_t *sq = q->q_syncq;
1088         clock_t ret;

1090         mutex_enter(SQLOCK(sq));
1091         /* callbacks are processed serially on each syncq */
1092         while (sq->sq_callbflags & SQ_CALLB_CANCEL_MASK) {
1093                 sq->sq_flags |= SQ_WANTWAKEUP;
1094                 cv_wait(&sq->sq_wait, SQLOCK(sq));
1095         }
1096         sq->sq_cancelid = (callbparams_id_t)id;
1097         sq->sq_callbflags = SQ_CANCEL_TOUT;
1098         if (sq->sq_flags & SQ_WANTWAKEUP) {
1099                 cv_broadcast(&sq->sq_wait);
1100                 sq->sq_flags &= ~SQ_WANTWAKEUP;
1101         }
1102         mutex_exit(SQLOCK(sq));
1103         ret = untimeout(id);
1104         mutex_enter(SQLOCK(sq));
1105         if (ret != -1) {
1106                 /* The wrapper was never called - need to free based on id */
1107                 callbparams_free_id(sq, (callbparams_id_t)id, SQ_CANCEL_TOUT);
1108         }
1109         if (sq->sq_callbflags & SQ_CALLB_BYPASSED) {
1110                 ret = 0;        /* this was how much time left */
1111         }
1112         sq->sq_callbflags = 0;
1113         if (sq->sq_flags & SQ_WANTWAKEUP) {
1114                 cv_broadcast(&sq->sq_wait);
1115                 sq->sq_flags &= ~SQ_WANTWAKEUP;
1116         }
1117         mutex_exit(SQLOCK(sq));
```



```
1118         return (ret);
1119 }


1122 void
1123 qunbufcall(queue_t *q, bufcall_id_t id)
1124 {
1125         syncq_t *sq = q->q_syncq;

1127         mutex_enter(SQLOCK(sq));
1128         /* callbacks are processed serially on each syncq */
1129         while (sq->sq_callbflags & SQ_CALLB_CANCEL_MASK) {
1130                 sq->sq_flags |= SQ_WANTWAKEUP;
1131                 cv_wait(&sq->sq_wait, SQLOCK(sq));
1132         }
1133         sq->sq_cancelid = (callbparams_id_t)id;
1134         sq->sq_callbflags = SQ_CANCEL_BUFCALL;
1135         if (sq->sq_flags & SQ_WANTWAKEUP) {
1136                 cv_broadcast(&sq->sq_wait);
1137                 sq->sq_flags &= ~SQ_WANTWAKEUP;
1138         }
1139         mutex_exit(SQLOCK(sq));
1140         unbufcall(id);
1141         mutex_enter(SQLOCK(sq));
1142         /*
1143          * No indication from unbufcall if the callback has already run.
1144          * Always attempt to free it.
1145          */
1146         callbparams_free_id(sq, (callbparams_id_t)id, SQ_CANCEL_BUFCALL);
1147         sq->sq_callbflags = 0;
1148         if (sq->sq_flags & SQ_WANTWAKEUP) {
1149                 cv_broadcast(&sq->sq_wait);
1150                 sq->sq_flags &= ~SQ_WANTWAKEUP;
1151         }
1152         mutex_exit(SQLOCK(sq));
1153 }

1155 /*
1156  * Associate the stream with an instance of the bottom driver.  This
1157  * function is called by APIs that establish or modify the hardware
1158  * association (ppa) of an open stream.  Two examples of such
1159  * post-open(9E) APIs are the dlpi(7p) DL_ATTACH_REQ message, and the
1160  * ndd(1M) "instance=" ioctl(2).  This interface may be called from a
1161  * stream driver's wput procedure and from within syncq perimeters,
1162  * so it can't block.
1163  *
1164  * The qassociate() "model" is that it should drive attach(9E), yet it
1165  * can't really do that because driving attach(9E) is a blocking
1166  * operation.  Instead, the qassociate() implementation has complex
1167  * dependencies on the implementation behavior of other parts of the
1168  * kernel to ensure all appropriate instances (ones that have not been
1169  * made inaccessible by DR) are attached at stream open() time, and
1170  * that they will not autodetach.  The code relies on the fact that an
1171  * open() of a stream that ends up using qassociate() always occurs on
1172  * a minor node created with CLONE_DEV.  The open() comes through
1173  * clnopen() and since clnopen() calls ddi_hold_installed_driver() we
1174  * attach all instances and mark them DN_NO_AUTODETACH (given
1175  * DN_DRIVER_HELD is maintained correctly).
1176  *
1177  * Since qassociate() can't really drive attach(9E), there are corner
1178  * cases where the compromise described above leads to qassociate()
1179  * returning failure.  This can happen when administrative functions
1180  * that cause detach(9E), such as "update_drv" or "modunload -i", are
1181  * performed on the driver between the time the stream was opened and
1182  * the time its hardware association was established.  Although this can
1183  * theoretically be an arbitrary amount of time, in practice the window
```

```
1184  * is usually quite small, since applications almost always issue their
1185  * hardware association request immediately after opening the stream,
1186  * and do not typically switch association while open.  When these
1187  * corner cases occur, and qassociate() finds the requested instance
1188  * detached, it will return failure.  This failure should be propagated
1189  * to the requesting administrative application using the appropriate
1190  * post-open(9E) API error mechanism.
1191  *
1192  * All qassociate() callers are expected to check for and gracefully handle
1193  * failure return, propagating errors back to the requesting administrative
1194  * application.
1195  */
1196 int
1197 qassociate(queue_t *q, int instance)
1198 {
1199         vnode_t *vp;
1200         major_t major;
1201         dev_info_t *dip;

1203         if (instance == -1) {
1204                 ddi_assoc_queue_with_devi(q, NULL);
1205                 return (0);
1206         }

1208         vp = STREAM(q)->sd_vnode;
1209         major = getmajor(vp->v_rdev);
1210         dip = ddi_hold_devi_by_instance(major, instance,
1211             E_DDI_HOLD_DEVI_NOATTACH);
1212         if (dip == NULL)
1213                 return (-1);

1215         ddi_assoc_queue_with_devi(q, dip);
1216         ddi_release_devi(dip);
1217         return (0);
1218 }

1220 /*
1221  * This routine is the SVR4MP 'replacement' for
1222  * hat_getkpfnum.  The only major difference is
1223  * the return value for illegal addresses - since
1224  * sunm_getkpfnum() and srmmu_getkpfnum() both
1225  * return '-1' for bogus mappings, we can (more or
1226  * less) return the value directly.
1227  */
1228 ppid_t
1229 kvtoppid(caddr_t addr)
1230 {
1231         return ((ppid_t)hat_getpfnum(kas.a_hat, addr));
1232 }

1234 /*
1235  * This is used to set the timeout value for cv_timed_wait() or
1236  * cv_timedwait_sig().
1237  */
1238 void
1239 time_to_wait(clock_t *now, clock_t time)
1240 {
1241         *now = ddi_get_lbolt() + time;
1242 }
```

```
**********************************************************
   38223 Wed Aug 19 07:25:20 2015
new/usr/src/uts/common/os/ddi_intr_irm.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 818 /*
 819  * Interrupt Pool Balancing
 820  */

 822 /*
 823  * irm_balance_thread()
 824  *
 825  *      One instance of this thread operates per each defined IRM pool.
 826  *      It does the initial activation of the pool, as well as balancing
 827  *      any requests that were queued up before the pool was active.
 828  *      Once active, it waits forever to service balance operations.
 829  */
 830 static void
 831 irm_balance_thread(ddi_irm_pool_t *pool_p)
 832 {
 833         clock_t         interval;

 835         DDI_INTR_IRMDBG((CE_CONT, "irm_balance_thread: pool_p %p\n",
 836             (void *)pool_p));

 838         /* Lock the pool */
 839         mutex_enter(&pool_p->ipool_lock);

 841         /* Perform initial balance if required */
 842         if (pool_p->ipool_reqno > pool_p->ipool_resno)
 843                 i_ddi_irm_balance(pool_p);

 845         /* Activate the pool */
 846         pool_p->ipool_flags |= DDI_IRM_FLAG_ACTIVE;

 848         /*
 849          * Main loop.
 850          * Iterate once first before wait on signal, in case there is signal
 851          * sent before this thread being created
 852          */
 853         for (;;) {

 855                 /* Compute the delay interval */
 856                 interval = drv_sectohz(irm_balance_delay);
 856                 interval = drv_usectohz(irm_balance_delay * 1000000);

 858                 /* Wait one interval, or until there are waiters */
 859                 if ((interval > 0) &&
 860                     !(pool_p->ipool_flags & DDI_IRM_FLAG_WAITERS) &&
 861                     !(pool_p->ipool_flags & DDI_IRM_FLAG_EXIT)) {
 862                         (void) cv_reltimedwait(&pool_p->ipool_cv,
 863                             &pool_p->ipool_lock, interval, TR_CLOCK_TICK);
 864                 }

 866                 /* Check if awakened to exit */
 867                 if (pool_p->ipool_flags & DDI_IRM_FLAG_EXIT) {
 868                         DDI_INTR_IRMDBG((CE_CONT,
 869                             "irm_balance_thread: exiting...\n"));
 870                         mutex_exit(&pool_p->ipool_lock);
 871                         thread_exit();
 872                 }

 874                 /* Balance the pool */
 875                 i_ddi_irm_balance(pool_p);
```



```
 877                 /* Notify waiters */
 878                 if (pool_p->ipool_flags & DDI_IRM_FLAG_WAITERS) {
 879                         cv_broadcast(&pool_p->ipool_cv);
 880                         pool_p->ipool_flags &= ~(DDI_IRM_FLAG_WAITERS);
 881                 }

 883                 /* Clear QUEUED condition */
 884                 pool_p->ipool_flags &= ~(DDI_IRM_FLAG_QUEUED);

 886                 /* Sleep until queued */
 887                 cv_wait(&pool_p->ipool_cv, &pool_p->ipool_lock);

 889                 DDI_INTR_IRMDBG((CE_CONT, "irm_balance_thread: signaled.\n"));
 890         }
 891 }
______unchanged_portion_omitted_
```

```
**********************************************************
   28685 Wed Aug 19 07:25:20 2015
new/usr/src/uts/common/os/devcache.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/note.h>
#include <sys/t_lock.h>
#include <sys/cmn_err.h>
#include <sys/instance.h>
#include <sys/conf.h>
#include <sys/stat.h>
#include <sys/ddi.h>
#include <sys/hwconf.h>
#include <sys/sunddi.h>
#include <sys/sunndi.h>
#include <sys/ddi_impldefs.h>
#include <sys/ndi_impldefs.h>
#include <sys/modctl.h>
#include <sys/dacf.h>
#include <sys/promif.h>
#include <sys/cpuvar.h>
#include <sys/pathname.h>
#include <sys/kobj.h>
#include <sys/devcache.h>
#include <sys/devcache_impl.h>
#include <sys/sysmacros.h>
#include <sys/varargs.h>
#include <sys/callb.h>

/*
 * This facility provides interfaces to clients to register,
 * read and update cache data in persisted backing store files,
 * usually in /etc/devices.  The data persisted through this
 * mechanism should be stateless data, functioning in the sense
 * of a cache.  Writes are performed by a background daemon
 * thread, permitting a client to schedule an update without
 * blocking, then continue updating the data state in
 * parallel.  The data is only locked by the daemon thread
 * to pack the data in preparation for the write.
 *
 * Data persisted through this mechanism should be capable
```



```
 * of being regenerated through normal system operation,
 * for example attaching all disk devices would cause all
 * devids to be registered for those devices.  By caching
 * a devid-device tuple, the system can operate in a
 * more optimal way, directly attaching the device mapped
 * to a devid, rather than burdensomely driving attach of
 * the entire device tree to discover a single device.
 *
 * Note that a client should only need to include
 * <sys/devcache.h> for the supported interfaces.
 *
 * The data per client is entirely within the control of
 * the client.  When reading, data unpacked from the backing
 * store should be inserted in the list.  The pointer to
 * the list can be retrieved via nvf_list().  When writing,
 * the data on the list is to be packed and returned to the
 * nvpdaemon as an nvlist.
 *
 * Obvious restrictions are imposed by the limits of the
 * nvlist format.  The data cannot be read or written
 * piecemeal, and large amounts of data aren't recommended.
 * However, nvlists do allow that data be named and typed
 * and can be size-of-int invariant, and the cached data
 * can be versioned conveniently.
 *
 * The registration involves two steps: a handle is
 * allocated by calling the registration function.
 * This sets up the data referenced by the handle and
 * initializes the lock.  Following registration, the
 * client must initialize the data list.  The list
 * interfaces require that the list element with offset
 * to the node link be provided.  The format of the
 * list element is under the control of the client.
 *
 * Locking: the address of the data list r/w lock provided
 * can be accessed with nvf_lock().  The lock must be held
 * as reader when traversing the list or checking state,
 * such as nvf_is_dirty().  The lock must be held as
 * writer when updating the list or marking it dirty.
 * The lock must not be held when waking the daemon.
 *
 * The data r/w lock is held as writer when the pack,
 * unpack and free list handlers are called.  The
 * lock should not be dropped and must be still held
 * upon return.  The client should also hold the lock
 * as reader when checking if the list is dirty, and
 * as writer when marking the list dirty or initiating
 * a read.
 *
 * The asynchronous nature of updates allows for the
 * possibility that the data may continue to be updated
 * once the daemon has been notified that an update is
 * desired.  The data only needs to be locked against
 * updates when packing the data into the form to be
 * written.  When the write of the packed data has
 * completed, the daemon will automatically reschedule
 * an update if the data was marked dirty after the
 * point at which it was packed.  Before beginning an
 * update, the daemon attempts to lock the data as
 * writer; if the writer lock is already held, it
 * backs off and retries later.  The model is to give
 * priority to the kernel processes generating the
 * data, and that the nature of the data is that
 * it does not change often, can be re-generated when
 * needed, so updates should not happen often and
 * can be delayed until the data stops changing.
```

```
 128  * The client may update the list or mark it dirty
 129  * any time it is able to acquire the lock as
 130  * writer first.
 131  *
 132  * A failed write will be retried after some delay,
 133  * in the hope that the cause of the error will be
 134  * transient, for example a filesystem with no space
 135  * available.  An update on a read-only filesystem
 136  * is failed silently and not retried; this would be
 137  * the case when booted off install media.
 138  *
 139  * There is no unregister mechanism as of yet, as it
 140  * hasn't been needed so far.
 141  */

 143 /*
 144  * Global list of files registered and updated by the nvpflush
 145  * daemon, protected by the nvf_cache_mutex.  While an
 146  * update is taking place, a file is temporarily moved to
 147  * the dirty list to avoid locking the primary list for
 148  * the duration of the update.
 149  */
 150 list_t          nvf_cache_files;
 151 list_t          nvf_dirty_files;
 152 kmutex_t        nvf_cache_mutex;


 155 /*
 156  * Allow some delay from an update of the data before flushing
 157  * to permit simultaneous updates of multiple changes.
 158  * Changes in the data are expected to be bursty, ie
 159  * reconfig or hot-plug of a new adapter.
 160  *
 161  * kfio_report_error (default 0)
 162  *      Set to 1 to enable some error messages related to low-level
 163  *      kernel file i/o operations.
 164  *
 165  * nvpflush_delay (default 10)
 166  *      The number of seconds after data is marked dirty before the
 167  *      flush daemon is triggered to flush the data.  A longer period
 168  *      of time permits more data updates per write.  Note that
 169  *      every update resets the timer so no repository write will
 170  *      occur while data is being updated continuously.
 171  *
 172  * nvpdaemon_idle_time (default 60)
 173  *      The number of seconds the daemon will sleep idle before exiting.
 174  *
 175  */
 176 #define NVPFLUSH_DELAY          10
 177 #define NVPDAEMON_IDLE_TIME     60

 179 #define TICKS_PER_SECOND        drv_sectohz(1)
 179 #define TICKS_PER_SECOND        (drv_usectohz(1000000))

 181 /*
 182  * Tunables
 183  */
 184 int kfio_report_error = 0;              /* kernel file i/o operations */
 185 int kfio_disable_read = 0;              /* disable all reads */
 186 int kfio_disable_write = 0;             /* disable all writes */

 188 int nvpflush_delay      = NVPFLUSH_DELAY;
 189 int nvpdaemon_idle_time = NVPDAEMON_IDLE_TIME;

 191 static timeout_id_t     nvpflush_id = 0;
 192 static int              nvpflush_timer_busy = 0;
```



```
 193 static int              nvpflush_daemon_active = 0;
 194 static kthread_t        *nvpflush_thr_id = 0;

 196 static int              do_nvpflush = 0;
 197 static int              nvpbusy = 0;
 198 static kmutex_t         nvpflush_lock;
 199 static kcondvar_t       nvpflush_cv;
 200 static kthread_id_t     nvpflush_thread;
 201 static clock_t          nvpticks;

 203 static void nvpflush_daemon(void);

 205 #ifdef  DEBUG
 206 int nvpdaemon_debug = 0;
 207 int kfio_debug = 0;
 208 #endif  /* DEBUG */

 210 extern int modrootloaded;
 211 extern void mdi_read_devices_files(void);
 212 extern void mdi_clean_vhcache(void);
 213 extern int sys_shutdown;

 215 /*
 216  * Initialize the overall cache file management
 217  */
 218 void
 219 i_ddi_devices_init(void)
 220 {
 221         list_create(&nvf_cache_files, sizeof (nvfd_t),
 222             offsetof(nvfd_t, nvf_link));
 223         list_create(&nvf_dirty_files, sizeof (nvfd_t),
 224             offsetof(nvfd_t, nvf_link));
 225         mutex_init(&nvf_cache_mutex, NULL, MUTEX_DEFAULT, NULL);
 226         retire_store_init();
 227         devid_cache_init();
 228 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  235623 Wed Aug 19 07:25:20 2015
new/usr/src/uts/common/os/devcfg.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


4270 /*
4271  * devtree_freeze() must be called before quiesce_devices() and reset_leaves()
4272  * during a normal system shutdown.  It attempts to ensure that there are no
4273  * outstanding attach or detach operations in progress when quiesce_devices() or
4274  * reset_leaves()is invoked.  It must be called before the system becomes
4275  * single-threaded because device attach and detach are multi-threaded
4276  * operations.  (note that during system shutdown the system doesn't actually
4277  * become single-thread since other threads still exist, but the shutdown thread
4278  * will disable preemption for itself, raise it's pil, and stop all the other
4279  * cpus in the system there by effectively making the system single-threaded.)
4280  */
4281 void
4282 devtree_freeze(void)
4283 {
4284         int delayed = 0;

4286         /* if we're panicing then the device tree isn't going to be changing */
4287         if (panicstr)
4288                 return;

4290         /* stop all dev_info state changes in the device tree */
4291         devinfo_freeze = gethrtime();

4293         /*
4294          * if we're not panicing and there are on-going attach or detach
4295          * operations, wait for up to 3 seconds for them to finish.  This
4296          * is a randomly chosen interval but this should be ok because:
4297          * - 3 seconds is very small relative to the deadman timer.
4298          * - normal attach and detach operations should be very quick.
4299          * - attach and detach operations are fairly rare.
4300          */
4301         while (!panicstr && atomic_add_long_nv(&devinfo_attach_detach, 0) &&
4302             (delayed < 3)) {
4303                 delayed += 1;

4305                 /* do a sleeping wait for one second */
4306                 ASSERT(!servicing_interrupt());
4307                 delay(drv_sectohz(1));
4307                 delay(drv_usectohz(MICROSEC));
4308         }
4309 }
______unchanged_portion_omitted_
```

****
30821 Wed Aug 19 07:25:21 2015
new/usr/src/uts/common/os/devid_cache.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
353 static int
354 e_devid_do_discovery(void)
355 {
356         ASSERT(mutex_owned(&devid_discovery_mutex));

358         if (i_ddi_io_initialized() == 0) {
359                 if (devid_discovery_boot > 0) {
360                         devid_discovery_boot--;
361                         return (1);
362                 }
363         } else {
364                 if (devid_discovery_postboot_always > 0)
365                         return (1);
366                 if (devid_discovery_postboot > 0) {
367                         devid_discovery_postboot--;
368                         return (1);
369                 }
370                 if (devid_discovery_secs > 0) {
371                         if ((ddi_get_lbolt() - devid_last_discovery) >
372                             drv_sectohz(devid_discovery_secs)) {
372                             drv_usectohz(devid_discovery_secs * MICROSEC)) {
373                                 return (1);
374                         }
375                 }
376         }

378         DEVID_LOG_DISC((CE_CONT, "devid_discovery: no discovery\n"));
379         return (0);
380 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  116054 Wed Aug 19 07:25:21 2015
new/usr/src/uts/common/os/modctl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3906 void
3907 mod_uninstall_daemon(void)
3908 {
3909         callb_cpr_t     cprinfo;
3910         clock_t         ticks;

3912         mod_aul_thread = curthread;

3914         CALLB_CPR_INIT(&cprinfo, &mod_uninstall_lock, callb_generic_cpr, "mud");
3915         for (;;) {
3916                 mutex_enter(&mod_uninstall_lock);
3917                 CALLB_CPR_SAFE_BEGIN(&cprinfo);
3918                 /*
3919                  * In DEBUG kernels, unheld drivers are uninstalled periodically
3920                  * every mod_uninstall_interval seconds.  Periodic uninstall can
3921                  * be disabled by setting mod_uninstall_interval to 0 which is
3922                  * the default for a non-DEBUG kernel.
3923                  */
3924                 if (mod_uninstall_interval) {
3925                         ticks = drv_sectohz(mod_uninstall_interval);
3925                         ticks = drv_usectohz(mod_uninstall_interval * 1000000);
3926                         (void) cv_reltimedwait(&mod_uninstall_cv,
3927                             &mod_uninstall_lock, ticks, TR_CLOCK_TICK);
3928                 } else {
3929                         cv_wait(&mod_uninstall_cv, &mod_uninstall_lock);
3930                 }
3931                 /*
3932                  * The whole daemon is safe for CPR except we don't want
3933                  * the daemon to run if FREEZE is issued and this daemon
3934                  * wakes up from the cv_wait above. In this case, it'll be
3935                  * blocked in CALLB_CPR_SAFE_END until THAW is issued.
3936                  *
3937                  * The reason of calling CALLB_CPR_SAFE_BEGIN twice is that
3938                  * mod_uninstall_lock is used to protect cprinfo and
3939                  * CALLB_CPR_SAFE_BEGIN assumes that this lock is held when
3940                  * called.
3941                  */
3942                 CALLB_CPR_SAFE_END(&cprinfo, &mod_uninstall_lock);
3943                 CALLB_CPR_SAFE_BEGIN(&cprinfo);
3944                 mutex_exit(&mod_uninstall_lock);
3945                 if ((modunload_disable_count == 0) &&
3946                     ((moddebug & MODDEBUG_NOAUTOUNLOAD) == 0)) {
3947                         mod_uninstall_all();
3948                 }
3949         }
3950 }
______unchanged_portion_omitted_
```

```
**********************************************************
  238235 Wed Aug 19 07:25:21 2015
new/usr/src/uts/common/os/sunmdi.c
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2014 Nexenta Systems Inc. All rights reserved.
 */

/*
 * Multipath driver interface (MDI) implementation; see mdi_impldefs.h for a
 * more detailed discussion of the overall mpxio architecture.
 *
 * Default locking order:
 *
 * _NOTE(LOCK_ORDER(mdi_mutex, mdi_vhci:vh_phci_mutex);
 * _NOTE(LOCK_ORDER(mdi_mutex, mdi_vhci:vh_client_mutex);
 * _NOTE(LOCK_ORDER(mdi_vhci:vh_phci_mutex, mdi_phci::ph_mutex);
 * _NOTE(LOCK_ORDER(mdi_vhci:vh_client_mutex, mdi_client::ct_mutex);
 * _NOTE(LOCK_ORDER(mdi_phci::ph_mutex mdi_pathinfo::pi_mutex))
 * _NOTE(LOCK_ORDER(mdi_phci::ph_mutex mdi_client::ct_mutex))
 * _NOTE(LOCK_ORDER(mdi_client::ct_mutex mdi_pathinfo::pi_mutex))
 */

#include <sys/note.h>
#include <sys/types.h>
#include <sys/varargs.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/kmem.h>
#include <sys/poll.h>
#include <sys/conf.h>
#include <sys/bootconf.h>
#include <sys/cmn_err.h>
#include <sys/stat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/ddipropdefs.h>
#include <sys/sunndi.h>
#include <sys/ndi_impldefs.h>
#include <sys/promif.h>
```



```
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/taskq.h>
#include <sys/epm.h>
#include <sys/sunpm.h>
#include <sys/modhash.h>
#include <sys/disp.h>
#include <sys/autoconf.h>
#include <sys/sysmacros.h>

#ifdef  DEBUG
#include <sys/debug.h>
int     mdi_debug = 1;
int     mdi_debug_logonly = 0;
#define MDI_DEBUG(dbglevel, pargs) if (mdi_debug >= (dbglevel)) i_mdi_log pargs
#define MDI_WARN        CE_WARN, __func__
#define MDI_NOTE        CE_NOTE, __func__
#define MDI_CONT        CE_CONT, __func__
static void i_mdi_log(int, const char *, dev_info_t *, const char *, ...);
#else   /* !DEBUG */
#define MDI_DEBUG(dbglevel, pargs)
#endif  /* DEBUG */
int     mdi_debug_consoleonly = 0;
int     mdi_delay = 3;

extern pri_t    minclsyspri;
extern int      modrootloaded;

/*
 * Global mutex:
 * Protects vHCI list and structure members.
 */
kmutex_t        mdi_mutex;

/*
 * Registered vHCI class driver lists
 */
int             mdi_vhci_count;
mdi_vhci_t      *mdi_vhci_head;
mdi_vhci_t      *mdi_vhci_tail;

/*
 * Client Hash Table size
 */
static int      mdi_client_table_size = CLIENT_HASH_TABLE_SIZE;

/*
 * taskq interface definitions
 */
#define MDI_TASKQ_N_THREADS     8
#define MDI_TASKQ_PRI           minclsyspri
#define MDI_TASKQ_MINALLOC      (4*mdi_taskq_n_threads)
#define MDI_TASKQ_MAXALLOC      (500*mdi_taskq_n_threads)

taskq_t                         *mdi_taskq;
static uint_t                   mdi_taskq_n_threads = MDI_TASKQ_N_THREADS;

#define TICKS_PER_SECOND        drv_sectohz(1)
#define TICKS_PER_SECOND        (drv_usectohz(1000000))

/*
 * The data should be "quiet" for this interval (in seconds) before the
 * vhci cached data is flushed to the disk.
 */
static int mdi_vhcache_flush_delay = 10;
```

```
127 /* number of seconds the vhcache flush daemon will sleep idle before exiting */
128 static int mdi_vhcache_flush_daemon_idle_time = 60;

130 /*
131  * MDI falls back to discovery of all paths when a bus_config_one fails.
132  * The following parameters can be used to tune this operation.
133  *
134  * mdi_path_discovery_boot
135  *      Number of times path discovery will be attempted during early boot.
136  *      Probably there is no reason to ever set this value to greater than one.
137  *
138  * mdi_path_discovery_postboot
139  *      Number of times path discovery will be attempted after early boot.
140  *      Set it to a minimum of two to allow for discovery of iscsi paths which
141  *      may happen very late during booting.
142  *
143  * mdi_path_discovery_interval
144  *      Minimum number of seconds MDI will wait between successive discovery
145  *      of all paths. Set it to -1 to disable discovery of all paths.
146  */
147 static int mdi_path_discovery_boot = 1;
148 static int mdi_path_discovery_postboot = 2;
149 static int mdi_path_discovery_interval = 10;

151 /*
152  * number of seconds the asynchronous configuration thread will sleep idle
153  * before exiting.
154  */
155 static int mdi_async_config_idle_time = 600;

157 static int mdi_bus_config_cache_hash_size = 256;

159 /* turns off multithreaded configuration for certain operations */
160 static int mdi_mtc_off = 0;

162 /*
163  * The "path" to a pathinfo node is identical to the /devices path to a
164  * devinfo node had the device been enumerated under a pHCI instead of
165  * a vHCI.  This pathinfo "path" is associated with a 'path_instance'.
166  * This association persists across create/delete of the pathinfo nodes,
167  * but not across reboot.
168  */
169 static uint_t           mdi_pathmap_instance = 1;       /* 0 -> any path */
170 static int              mdi_pathmap_hash_size = 256;
171 static kmutex_t         mdi_pathmap_mutex;
172 static mod_hash_t       *mdi_pathmap_bypath;            /* "path"->instance */
173 static mod_hash_t       *mdi_pathmap_byinstance;        /* instance->"path" */
174 static mod_hash_t       *mdi_pathmap_sbyinstance;       /* inst->shortpath */

176 /*
177  * MDI component property name/value string definitions
178  */
179 const char              *mdi_component_prop = "mpxio-component";
180 const char              *mdi_component_prop_vhci = "vhci";
181 const char              *mdi_component_prop_phci = "phci";
182 const char              *mdi_component_prop_client = "client";

184 /*
185  * MDI client global unique identifier property name
186  */
187 const char              *mdi_client_guid_prop = "client-guid";

189 /*
190  * MDI client load balancing property name/value string definitions
191  */
192 const char              *mdi_load_balance = "load-balance";
```



```
193 const char              *mdi_load_balance_none = "none";
194 const char              *mdi_load_balance_rr = "round-robin";
195 const char              *mdi_load_balance_lba = "logical-block";

197 /*
198  * Obsolete vHCI class definition; to be removed after Leadville update
199  */
200 const char *mdi_vhci_class_scsi = MDI_HCI_CLASS_SCSI;

202 static char vhci_greeting[] =
203         "\tThere already exists one vHCI driver for class %s\n"
204         "\tOnly one vHCI driver for each class is allowed\n";

206 /*
207  * Static function prototypes
208  */
209 static int              i_mdi_phci_offline(dev_info_t *, uint_t);
210 static int              i_mdi_client_offline(dev_info_t *, uint_t);
211 static int              i_mdi_phci_pre_detach(dev_info_t *, ddi_detach_cmd_t);
212 static void             i_mdi_phci_post_detach(dev_info_t *,
213                             ddi_detach_cmd_t, int);
214 static int              i_mdi_client_pre_detach(dev_info_t *,
215                             ddi_detach_cmd_t);
216 static void             i_mdi_client_post_detach(dev_info_t *,
217                             ddi_detach_cmd_t, int);
218 static void             i_mdi_pm_hold_pip(mdi_pathinfo_t *);
219 static void             i_mdi_pm_rele_pip(mdi_pathinfo_t *);
220 static int              i_mdi_lba_lb(mdi_client_t *ct,
221                             mdi_pathinfo_t **ret_pip, struct buf *buf);
222 static void             i_mdi_pm_hold_client(mdi_client_t *, int);
223 static void             i_mdi_pm_rele_client(mdi_client_t *, int);
224 static void             i_mdi_pm_reset_client(mdi_client_t *);
225 static int              i_mdi_power_all_phci(mdi_client_t *);
226 static void             i_mdi_log_sysevent(dev_info_t *, char *, char *);


229 /*
230  * Internal mdi_pathinfo node functions
231  */
232 static void             i_mdi_pi_kstat_destroy(mdi_pathinfo_t *);

234 static mdi_vhci_t       *i_mdi_vhci_class2vhci(char *);
235 static mdi_vhci_t       *i_devi_get_vhci(dev_info_t *);
236 static mdi_phci_t       *i_devi_get_phci(dev_info_t *);
237 static void             i_mdi_phci_lock(mdi_phci_t *, mdi_pathinfo_t *);
238 static void             i_mdi_phci_unlock(mdi_phci_t *);
239 static mdi_pathinfo_t   *i_mdi_pi_alloc(mdi_phci_t *, char *, mdi_client_t *);
240 static void             i_mdi_phci_add_path(mdi_phci_t *, mdi_pathinfo_t *);
241 static void             i_mdi_client_add_path(mdi_client_t *, mdi_pathinfo_t *);
242 static void             i_mdi_pi_free(mdi_phci_t *ph, mdi_pathinfo_t *,
243                             mdi_client_t *);
244 static void             i_mdi_phci_remove_path(mdi_phci_t *, mdi_pathinfo_t *);
245 static void             i_mdi_client_remove_path(mdi_client_t *,
246                             mdi_pathinfo_t *);

248 static int              i_mdi_pi_state_change(mdi_pathinfo_t *,
249                             mdi_pathinfo_state_t, int);
250 static int              i_mdi_pi_offline(mdi_pathinfo_t *, int);
251 static dev_info_t       *i_mdi_devinfo_create(mdi_vhci_t *, char *, char *,
252                             char **, int);
253 static dev_info_t       *i_mdi_devinfo_find(mdi_vhci_t *, char *, char *);
254 static int              i_mdi_devinfo_remove(dev_info_t *, dev_info_t *, int);
255 static int              i_mdi_is_child_present(dev_info_t *, dev_info_t *);
256 static mdi_client_t     *i_mdi_client_alloc(mdi_vhci_t *, char *, char *);
257 static void             i_mdi_client_enlist_table(mdi_vhci_t *, mdi_client_t *);
258 static void             i_mdi_client_delist_table(mdi_vhci_t *, mdi_client_t *);
```

```
259 static mdi_client_t     *i_mdi_client_find(mdi_vhci_t *, char *, char *);
260 static void              i_mdi_client_update_state(mdi_client_t *);
261 static int               i_mdi_client_compute_state(mdi_client_t *,
262                              mdi_phci_t *);
263 static void              i_mdi_client_lock(mdi_client_t *, mdi_pathinfo_t *);
264 static void              i_mdi_client_unlock(mdi_client_t *);
265 static int               i_mdi_client_free(mdi_vhci_t *, mdi_client_t *);
266 static mdi_client_t     *i_devi_get_client(dev_info_t *);
267 /*
268  * NOTE: this will be removed once the NWS files are changed to use the new
269  * mdi_{enable,disable}_path interfaces
270  */
271 static int               i_mdi_pi_enable_disable(dev_info_t *, dev_info_t *,
272                                  int, int);
273 static mdi_pathinfo_t   *i_mdi_enable_disable_path(mdi_pathinfo_t *pip,
274                                  mdi_vhci_t *vh, int flags, int op);
275 /*
276  * Failover related function prototypes
277  */
278 static int               i_mdi_failover(void *);

280 /*
281  * misc internal functions
282  */
283 static int               i_mdi_get_hash_key(char *);
284 static int               i_map_nvlist_error_to_mdi(int);
285 static void              i_mdi_report_path_state(mdi_client_t *,
286                              mdi_pathinfo_t *);

288 static void              setup_vhci_cache(mdi_vhci_t *);
289 static int               destroy_vhci_cache(mdi_vhci_t *);
290 static int               stop_vhcache_async_threads(mdi_vhci_config_t *);
291 static boolean_t         stop_vhcache_flush_thread(void *, int);
292 static void              free_string_array(char **, int);
293 static void              free_vhcache_phci(mdi_vhcache_phci_t *);
294 static void              free_vhcache_pathinfo(mdi_vhcache_pathinfo_t *);
295 static void              free_vhcache_client(mdi_vhcache_client_t *);
296 static int               mainnvl_to_vhcache(mdi_vhci_cache_t *, nvlist_t *);
297 static nvlist_t          *vhcache_to_mainnvl(mdi_vhci_cache_t *);
298 static void              vhcache_phci_add(mdi_vhci_config_t *, mdi_phci_t *);
299 static void              vhcache_phci_remove(mdi_vhci_config_t *, mdi_phci_t *);
300 static void              vhcache_pi_add(mdi_vhci_config_t *,
301                              struct mdi_pathinfo *);
302 static void              vhcache_pi_remove(mdi_vhci_config_t *,
303                              struct mdi_pathinfo *);
304 static void              free_phclient_path_list(mdi_phys_path_t *);
305 static void              sort_vhcache_paths(mdi_vhcache_client_t *);
306 static int               flush_vhcache(mdi_vhci_config_t *, int);
307 static void              vhcache_dirty(mdi_vhci_config_t *);
308 static void              free_async_client_config(mdi_async_client_config_t *);
309 static void              single_threaded_vhconfig_enter(mdi_vhci_config_t *);
310 static void              single_threaded_vhconfig_exit(mdi_vhci_config_t *);
311 static nvlist_t          *read_on_disk_vhci_cache(char *);
312 extern int               fread_nvlist(char *, nvlist_t **);
313 extern int               fwrite_nvlist(char *, nvlist_t *);

315 /* called once when first vhci registers with mdi */
316 static void
317 i_mdi_init()
318 {
319         static int initialized = 0;

321         if (initialized)
322                 return;
323         initialized = 1;
```



```
325         mutex_init(&mdi_mutex, NULL, MUTEX_DEFAULT, NULL);

327         /* Create our taskq resources */
328         mdi_taskq = taskq_create("mdi_taskq", mdi_taskq_n_threads,
329             MDI_TASKQ_PRI, MDI_TASKQ_MINALLOC, MDI_TASKQ_MAXALLOC,
330             TASKQ_PREPOPULATE | TASKQ_CPR_SAFE);
331         ASSERT(mdi_taskq != NULL);      /* taskq_create never fails */

333         /* Allocate ['path_instance' <-> "path"] maps */
334         mutex_init(&mdi_pathmap_mutex, NULL, MUTEX_DRIVER, NULL);
335         mdi_pathmap_bypath = mod_hash_create_strhash(
336             "mdi_pathmap_bypath", mdi_pathmap_hash_size,
337             mod_hash_null_valdtor);
338         mdi_pathmap_byinstance = mod_hash_create_idhash(
339             "mdi_pathmap_byinstance", mdi_pathmap_hash_size,
340             mod_hash_null_valdtor);
341         mdi_pathmap_sbyinstance = mod_hash_create_idhash(
342             "mdi_pathmap_sbyinstance", mdi_pathmap_hash_size,
343             mod_hash_null_valdtor);
344 }
```

*unchanged_portion_omitted_*

```
3120 /*
3121  * mdi_pi_free():
3122  *              Free the mdi_pathinfo node and also client device node if this
3123  *              is the last path to the device
3124  * Return Values:
3125  *              MDI_SUCCESS
3126  *              MDI_FAILURE
3127  *              MDI_BUSY
3128  */
3129 /*ARGSUSED*/
3130 int
3131 mdi_pi_free(mdi_pathinfo_t *pip, int flags)
3132 {
3133         int             rv;
3134         mdi_vhci_t      *vh;
3135         mdi_phci_t      *ph;
3136         mdi_client_t    *ct;
3137         int             (*f)();
3138         int             client_held = 0;

3140         MDI_PI_LOCK(pip);
3141         ph = MDI_PI(pip)->pi_phci;
3142         ASSERT(ph != NULL);
3143         if (ph == NULL) {
3144                 /*
3145                  * Invalid pHCI device, return failure
3146                  */
3147                 MDI_DEBUG(1, (MDI_WARN, NULL,
3148                     "!invalid pHCI: pip %s %p",
3149                     mdi_pi_spathname(pip), (void *)pip));
3150                 MDI_PI_UNLOCK(pip);
3151                 return (MDI_FAILURE);
3152         }

3154         vh = ph->ph_vhci;
3155         ASSERT(vh != NULL);
3156         if (vh == NULL) {
3157                 /* Invalid pHCI device, return failure */
3158                 MDI_DEBUG(1, (MDI_WARN, ph->ph_dip,
3159                     "!invalid vHCI: pip %s %p",
3160                     mdi_pi_spathname(pip), (void *)pip));
3161                 MDI_PI_UNLOCK(pip);
3162                 return (MDI_FAILURE);
3163         }
```

```
3165         ct = MDI_PI(pip)->pi_client;
3166         ASSERT(ct != NULL);
3167         if (ct == NULL) {
3168                 /*
3169                  * Invalid Client device, return failure
3170                  */
3171                 MDI_DEBUG(1, (MDI_WARN, ph->ph_dip,
3172                     "!invalid client: pip %s %p",
3173                     mdi_pi_spathname(pip), (void *)pip));
3174                 MDI_PI_UNLOCK(pip);
3175                 return (MDI_FAILURE);
3176         }

3178         /*
3179          * Check to see for busy condition.  A mdi_pathinfo can only be freed
3180          * if the node state is either offline or init and the reference count
3181          * is zero.
3182          */
3183         if (!(MDI_PI_IS_OFFLINE(pip) || MDI_PI_IS_INIT(pip) ||
3184             MDI_PI_IS_INITING(pip))) {
3185                 /*
3186                  * Node is busy
3187                  */
3188                 MDI_DEBUG(1, (MDI_WARN, ct->ct_dip,
3189                     "!busy: pip %s %p", mdi_pi_spathname(pip), (void *)pip));
3190                 MDI_PI_UNLOCK(pip);
3191                 return (MDI_BUSY);
3192         }

3194         while (MDI_PI(pip)->pi_ref_cnt != 0) {
3195                 /*
3196                  * Give a chance for pending I/Os to complete.
3197                  */
3198                 MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3199                     "!%d cmds still pending on path: %s %p",
3200                     MDI_PI(pip)->pi_ref_cnt,
3201                     mdi_pi_spathname(pip), (void *)pip));
3202                 if (cv_reltimedwait(&MDI_PI(pip)->pi_ref_cv,
3203                     &MDI_PI(pip)->pi_mutex, drv_sectohz(60),
3203                     &MDI_PI(pip)->pi_mutex, drv_usectohz(60 * 1000000),
3204                     TR_CLOCK_TICK) == -1) {
3205                         /*
3206                          * The timeout time reached without ref_cnt being zero
3207                          * being signaled.
3208                          */
3209                         MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3210                             "!Timeout reached on path %s %p without the cond",
3211                             mdi_pi_spathname(pip), (void *)pip));
3212                         MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3213                             "!%d cmds still pending on path %s %p",
3214                             MDI_PI(pip)->pi_ref_cnt,
3215                             mdi_pi_spathname(pip), (void *)pip));
3216                         MDI_PI_UNLOCK(pip);
3217                         return (MDI_BUSY);
3218                 }
3219         }
3220         if (MDI_PI(pip)->pi_pm_held) {
3221                 client_held = 1;
3222         }
3223         MDI_PI_UNLOCK(pip);

3225         vhcache_pi_remove(vh->vh_config, MDI_PI(pip));

3227         MDI_CLIENT_LOCK(ct);
```



```
3229         /* Prevent further failovers till MDI_VHCI_CLIENT_LOCK is held */
3230         MDI_CLIENT_SET_PATH_FREE_IN_PROGRESS(ct);

3232         /*
3233          * Wait till failover is complete before removing this node.
3234          */
3235         while (MDI_CLIENT_IS_FAILOVER_IN_PROGRESS(ct))
3236                 cv_wait(&ct->ct_failover_cv, &ct->ct_mutex);

3238         MDI_CLIENT_UNLOCK(ct);
3239         MDI_VHCI_CLIENT_LOCK(vh);
3240         MDI_CLIENT_LOCK(ct);
3241         MDI_CLIENT_CLEAR_PATH_FREE_IN_PROGRESS(ct);

3243         if (!MDI_PI_IS_INITING(pip)) {
3244                 f = vh->vh_ops->vo_pi_uninit;
3245                 if (f != NULL) {
3246                         rv = (*f)(vh->vh_dip, pip, 0);
3247                 }
3248         } else
3249                 rv = MDI_SUCCESS;

3251         /*
3252          * If vo_pi_uninit() completed successfully.
3253          */
3254         if (rv == MDI_SUCCESS) {
3255                 if (client_held) {
3256                         MDI_DEBUG(4, (MDI_NOTE, ct->ct_dip,
3257                             "i_mdi_pm_rele_client\n"));
3258                         i_mdi_pm_rele_client(ct, 1);
3259                 }
3260                 i_mdi_pi_free(ph, pip, ct);
3261                 if (ct->ct_path_count == 0) {
3262                         /*
3263                          * Client lost its last path.
3264                          * Clean up the client device
3265                          */
3266                         MDI_CLIENT_UNLOCK(ct);
3267                         (void) i_mdi_client_free(ct->ct_vhci, ct);
3268                         MDI_VHCI_CLIENT_UNLOCK(vh);
3269                         return (rv);
3270                 }
3271         }
3272         MDI_CLIENT_UNLOCK(ct);
3273         MDI_VHCI_CLIENT_UNLOCK(vh);

3275         if (rv == MDI_FAILURE)
3276                 vhcache_pi_add(vh->vh_config, MDI_PI(pip));

3278         return (rv);
3279 }
```

unchanged_portion_omitted_

```
3843 /*
3844  * i_mdi_pi_offline():
3845  *              Offline a mdi_pathinfo node and call the vHCI driver's callback
3846  */
3847 static int
3848 i_mdi_pi_offline(mdi_pathinfo_t *pip, int flags)
3849 {
3850         dev_info_t      *vdip = NULL;
3851         mdi_vhci_t      *vh = NULL;
3852         mdi_client_t    *ct = NULL;
3853         int             (*f)();
3854         int             rv;
```

```
3856         MDI_PI_LOCK(pip);
3857         ct = MDI_PI(pip)->pi_client;
3858         ASSERT(ct != NULL);

3860         while (MDI_PI(pip)->pi_ref_cnt != 0) {
3861                 /*
3862                  * Give a chance for pending I/Os to complete.
3863                  */
3864                 MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3865                     "!%d cmds still pending on path %s %p",
3866                     MDI_PI(pip)->pi_ref_cnt, mdi_pi_spathname(pip),
3867                     (void *)pip));
3868                 if (cv_reltimedwait(&MDI_PI(pip)->pi_ref_cv,
3869                     &MDI_PI(pip)->pi_mutex, drv_sectohz(60),
3869                     &MDI_PI(pip)->pi_mutex, drv_usectohz(60 * 1000000),
3870                     TR_CLOCK_TICK) == -1) {
3871                         /*
3872                          * The timeout time reached without ref_cnt being zero
3873                          * being signaled.
3874                          */
3875                         MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3876                             "!Timeout reached on path %s %p without the cond",
3877                             mdi_pi_spathname(pip), (void *)pip));
3878                         MDI_DEBUG(1, (MDI_NOTE, ct->ct_dip,
3879                             "!%d cmds still pending on path %s %p",
3880                             MDI_PI(pip)->pi_ref_cnt,
3881                             mdi_pi_spathname(pip), (void *)pip));
3882                 }
3883         }
3884         vh = ct->ct_vhci;
3885         vdip = vh->vh_dip;

3887         /*
3888          * Notify vHCI that has registered this event
3889          */
3890         ASSERT(vh->vh_ops);
3891         f = vh->vh_ops->vo_pi_state_change;

3893         if (f != NULL) {
3894                 MDI_PI_UNLOCK(pip);
3895                 if ((rv = (*f)(vdip, pip, MDI_PATHINFO_STATE_OFFLINE, 0,
3896                     flags)) != MDI_SUCCESS) {
3897                         MDI_DEBUG(1, (MDI_WARN, ct->ct_dip,
3898                             "!vo_path_offline failed: vdip %s%d %p: path %s %p",
3899                             ddi_driver_name(vdip), ddi_get_instance(vdip),
3900                             (void *)vdip, mdi_pi_spathname(pip), (void *)pip));
3901                 }
3902                 MDI_PI_LOCK(pip);
3903         }

3905         /*
3906          * Set the mdi_pathinfo node state and clear the transient condition
3907          */
3908         MDI_PI_SET_OFFLINE(pip);
3909         cv_broadcast(&MDI_PI(pip)->pi_state_cv);
3910         MDI_PI_UNLOCK(pip);

3912         MDI_CLIENT_LOCK(ct);
3913         if (rv == MDI_SUCCESS) {
3914                 if (ct->ct_unstable == 0) {
3915                         dev_info_t      *cdip = ct->ct_dip;

3917                         /*
3918                          * Onlining the mdi_pathinfo node will impact the
3919                          * client state Update the client and dev_info node
3920                          * state accordingly
```



```
3921                          */
3922                         i_mdi_client_update_state(ct);
3923                         rv = NDI_SUCCESS;
3924                         if (MDI_CLIENT_STATE(ct) == MDI_CLIENT_STATE_FAILED) {
3925                                 if (cdip &&
3926                                     (i_ddi_node_state(cdip) >=
3927                                     DS_INITIALIZED)) {
3928                                         MDI_CLIENT_UNLOCK(ct);
3929                                         rv = ndi_devi_offline(cdip,
3930                                             NDI_DEVFS_CLEAN);
3931                                         MDI_CLIENT_LOCK(ct);
3932                                         if (rv != NDI_SUCCESS) {
3933                                                 /*
3934                                                  * ndi_devi_offline failed.
3935                                                  * Reset client flags to
3936                                                  * online.
3937                                                  */
3938                                                 MDI_DEBUG(4, (MDI_WARN, cdip,
3939                                                     "ndi_devi_offline failed: "
3940                                                     "error %x", rv));
3941                                                 MDI_CLIENT_SET_ONLINE(ct);
3942                                         }
3943                                 }
3944                         }
3945                         /*
3946                          * Convert to MDI error code
3947                          */
3948                         switch (rv) {
3949                         case NDI_SUCCESS:
3950                                 rv = MDI_SUCCESS;
3951                                 break;
3952                         case NDI_BUSY:
3953                                 rv = MDI_BUSY;
3954                                 break;
3955                         default:
3956                                 rv = MDI_FAILURE;
3957                                 break;
3958                         }
3959                 }
3960                 MDI_CLIENT_SET_REPORT_DEV_NEEDED(ct);
3961                 i_mdi_report_path_state(ct, pip);
3962         }

3964         MDI_CLIENT_UNLOCK(ct);

3966         /*
3967          * Change in the mdi_pathinfo node state will impact the client state
3968          */
3969         MDI_DEBUG(2, (MDI_NOTE, ct->ct_dip,
3970             "ct = %p pip = %p", (void *)ct, (void *)pip));
3971         return (rv);
3972 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  105218 Wed Aug 19 07:25:21 2015
new/usr/src/uts/common/rpc/clnt_cots.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

765 static int clnt_cots_pulls;
766 #define RM_HDR_SIZE     4       /* record mark header size */

768 /*
769  * Call remote procedure.
770  */
771 static enum clnt_stat
772 clnt_cots_kcallit(CLIENT *h, rpcproc_t procnum, xdrproc_t xdr_args,
773     caddr_t argsp, xdrproc_t xdr_results, caddr_t resultsp, struct timeval wait)
774 {
775         /* LINTED pointer alignment */
776         cku_private_t *p = htop(h);
777         calllist_t *call = &p->cku_call;
778         XDR *xdrs;
779         struct rpc_msg reply_msg;
780         mblk_t *mp;
781 #ifdef  RPCDEBUG
782         clock_t time_sent;
783 #endif
784         struct netbuf *retryaddr;
785         struct cm_xprt *cm_entry = NULL;
786         queue_t *wq;
787         int len, waitsecs, max_waitsecs;
788         int mpsize;
789         int refreshes = REFRESHES;
790         int interrupted;
791         int tidu_size;
792         enum clnt_stat status;
793         struct timeval cwait;
794         bool_t delay_first = FALSE;
795         clock_t ticks, now;

797         RPCLOG(2, "clnt_cots_kcallit, procnum %u\n", procnum);
798         COTSRCSTAT_INCR(p->cku_stats, rccalls);

800         RPCLOG(2, "clnt_cots_kcallit: wait.tv_sec: %ld\n", wait.tv_sec);
801         RPCLOG(2, "clnt_cots_kcallit: wait.tv_usec: %ld\n", wait.tv_usec);
802         /*
803          * Bug ID 1240234:
804          * Look out for zero length timeouts. We don't want to
805          * wait zero seconds for a connection to be established.
806          */
807         if (wait.tv_sec < clnt_cots_min_conntout) {
808                 cwait.tv_sec = clnt_cots_min_conntout;
809                 cwait.tv_usec = 0;
810                 RPCLOG(8, "clnt_cots_kcallit: wait.tv_sec (%ld) too low,",
811                     wait.tv_sec);
812                 RPCLOG(8, " setting to: %d\n", clnt_cots_min_conntout);
813         } else {
814                 cwait = wait;
815         }

817 call_again:
818         if (cm_entry) {
819                 connmgr_release(cm_entry);
820                 cm_entry = NULL;
821         }

823         mp = NULL;
```



```
825         /*
826          * If the call is not a retry, allocate a new xid and cache it
827          * for future retries.
828          * Bug ID 1246045:
829          * Treat call as a retry for purposes of binding the source
830          * port only if we actually attempted to send anything on
831          * the previous call.
832          */
833         if (p->cku_xid == 0) {
834                 p->cku_xid = alloc_xid();
835                 call->call_zoneid = rpc_zoneid();

837                 /*
838                  * We need to ASSERT here that our xid != 0 because this
839                  * determines whether or not our call record gets placed on
840                  * the hash table or the linked list.  By design, we mandate
841                  * that RPC calls over cots must have xid's != 0, so we can
842                  * ensure proper management of the hash table.
843                  */
844                 ASSERT(p->cku_xid != 0);

846                 retryaddr = NULL;
847                 p->cku_flags &= ~CKU_SENT;

849                 if (p->cku_flags & CKU_ONQUEUE) {
850                         RPCLOG(8, "clnt_cots_kcallit: new call, dequeuing old"
851                             " one (%p)\n", (void *)call);
852                         call_table_remove(call);
853                         p->cku_flags &= ~CKU_ONQUEUE;
854                         RPCLOG(64, "clnt_cots_kcallit: removing call from "
855                             "dispatch list because xid was zero (now 0x%x)\n",
856                             p->cku_xid);
857                 }

859                 if (call->call_reply != NULL) {
860                         freemsg(call->call_reply);
861                         call->call_reply = NULL;
862                 }
863         } else if (p->cku_srcaddr.buf == NULL || p->cku_srcaddr.len == 0) {
864                 retryaddr = NULL;

866         } else if (p->cku_flags & CKU_SENT) {
867                 retryaddr = &p->cku_srcaddr;

869         } else {
870                 /*
871                  * Bug ID 1246045: Nothing was sent, so set retryaddr to
872                  * NULL and let connmgr_get() bind to any source port it
873                  * can get.
874                  */
875                 retryaddr = NULL;
876         }

878         RPCLOG(64, "clnt_cots_kcallit: xid = 0x%x", p->cku_xid);
879         RPCLOG(64, " flags = 0x%x\n", p->cku_flags);

881         p->cku_err.re_status = RPC_TIMEDOUT;
882         p->cku_err.re_errno = p->cku_err.re_terrno = 0;

884         cm_entry = connmgr_wrapget(retryaddr, &cwait, p);

886         if (cm_entry == NULL) {
887                 RPCLOG(1, "clnt_cots_kcallit: can't connect status %s\n",
888                     clnt_sperrno(p->cku_err.re_status));
```

```
890                 /*
891                  * The reasons why we fail to create a connection are
892                  * varied. In most cases we don't want the caller to
893                  * immediately retry. This could have one or more
894                  * bad effects. This includes flooding the net with
895                  * connect requests to ports with no listener; a hard
896                  * kernel loop due to all the "reserved" TCP ports being
897                  * in use.
898                  */
899                 delay_first = TRUE;
900
901                 /*
902                  * Even if we end up returning EINTR, we still count a
903                  * a "can't connect", because the connection manager
904                  * might have been committed to waiting for or timing out on
905                  * a connection.
906                  */
907                 COTSRCSTAT_INCR(p->cku_stats, rccantconn);
908                 switch (p->cku_err.re_status) {
909                 case RPC_INTR:
910                         p->cku_err.re_errno = EINTR;
911
912                         /*
913                          * No need to delay because a UNIX signal(2)
914                          * interrupted us. The caller likely won't
915                          * retry the CLNT_CALL() and even if it does,
916                          * we assume the caller knows what it is doing.
917                          */
918                         delay_first = FALSE;
919                         break;
920
921                 case RPC_TIMEDOUT:
922                         p->cku_err.re_errno = ETIMEDOUT;
923
924                         /*
925                          * No need to delay because timed out already
926                          * on the connection request and assume that the
927                          * transport time out is longer than our minimum
928                          * timeout, or least not too much smaller.
929                          */
930                         delay_first = FALSE;
931                         break;
932
933                 case RPC_SYSTEMERROR:
934                 case RPC_TLIERROR:
935                         /*
936                          * We want to delay here because a transient
937                          * system error has a better chance of going away
938                          * if we delay a bit. If it's not transient, then
939                          * we don't want end up in a hard kernel loop
940                          * due to retries.
941                          */
942                         ASSERT(p->cku_err.re_errno != 0);
943                         break;
944
945
946                 case RPC_CANTCONNECT:
947                         /*
948                          * RPC_CANTCONNECT is set on T_ERROR_ACK which
949                          * implies some error down in the TCP layer or
950                          * below. If cku_nodelayonerror is set then we
951                          * assume the caller knows not to try too hard.
952                          */
953                         RPCLOG0(8, "clnt_cots_kcallit: connection failed,");
954                         RPCLOG0(8, " re_status=RPC_CANTCONNECT,");
955                         RPCLOG(8, " re_errno=%d,", p->cku_err.re_errno);
```



```
956                         RPCLOG(8, " cku_nodelayonerr=%d", p->cku_nodelayonerr);
957                         if (p->cku_nodelayonerr == TRUE)
958                                 delay_first = FALSE;
959
960                         p->cku_err.re_errno = EIO;
961
962                         break;
963
964                 case RPC_XPRTFAILED:
965                         /*
966                          * We want to delay here because we likely
967                          * got a refused connection.
968                          */
969                         if (p->cku_err.re_errno == 0)
970                                 p->cku_err.re_errno = EIO;
971
972                         RPCLOG(1, "clnt_cots_kcallit: transport failed: %d\n",
973                             p->cku_err.re_errno);
974
975                         break;
976
977                 default:
978                         /*
979                          * We delay here because it is better to err
980                          * on the side of caution. If we got here then
981                          * status could have been RPC_SUCCESS, but we
982                          * know that we did not get a connection, so
983                          * force the rpc status to RPC_CANTCONNECT.
984                          */
985                         p->cku_err.re_status = RPC_CANTCONNECT;
986                         p->cku_err.re_errno = EIO;
987                         break;
988                 }
989                 if (delay_first == TRUE)
990                         ticks = drv_sectohz(clnt_cots_min_tout);
990                         ticks = clnt_cots_min_tout * drv_usectohz(1000000);
991                 goto cots_done;
992         }
993
994         /*
995          * If we've never sent any request on this connection (send count
996          * is zero, or the connection has been reset), cache the
997          * the connection's create time and send a request (possibly a retry)
998          */
999         if ((p->cku_flags & CKU_SENT) == 0 ||
1000             p->cku_ctime != cm_entry->x_ctime) {
1001                 p->cku_ctime = cm_entry->x_ctime;
1002
1003         } else if ((p->cku_flags & CKU_SENT) && (p->cku_flags & CKU_ONQUEUE) &&
1004             (call->call_reply != NULL ||
1005             p->cku_recv_attempts < clnt_cots_maxrecv)) {
1006
1007                 /*
1008                  * If we've sent a request and our call is on the dispatch
1009                  * queue and we haven't made too many receive attempts, then
1010                  * don't re-send, just receive.
1011                  */
1012                 p->cku_recv_attempts++;
1013                 goto read_again;
1014         }
1015
1016         /*
1017          * Now we create the RPC request in a STREAMS message.  We have to do
1018          * this after the call to connmgr_get so that we have the correct
1019          * TIDU size for the transport.
1020          */
```

```
1021         tidu_size = cm_entry->x_tidu_size;
1022         len = MSG_OFFSET + MAX(tidu_size, RM_HDR_SIZE + WIRE_HDR_SIZE);

1024         while ((mp = allocb(len, BPRI_MED)) == NULL) {
1025                 if (strwaitbuf(len, BPRI_MED)) {
1026                         p->cku_err.re_status = RPC_SYSTEMERROR;
1027                         p->cku_err.re_errno = ENOSR;
1028                         COTSRCSTAT_INCR(p->cku_stats, rcnomem);
1029                         goto cots_done;
1030                 }
1031         }
1032         xdrs = &p->cku_outxdr;
1033         xdrmblk_init(xdrs, mp, XDR_ENCODE, tidu_size);
1034         mpsize = MBLKSIZE(mp);
1035         ASSERT(mpsize >= len);
1036         ASSERT(mp->b_rptr == mp->b_datap->db_base);

1038         /*
1039          * If the size of mblk is not appreciably larger than what we
1040          * asked, then resize the mblk to exactly len bytes. The reason for
1041          * this: suppose len is 1600 bytes, the tidu is 1460 bytes
1042          * (from TCP over ethernet), and the arguments to the RPC require
1043          * 2800 bytes. Ideally we want the protocol to render two
1044          * ~1400 byte segments over the wire. However if allocb() gives us a 2k
1045          * mblk, and we allocate a second mblk for the remainder, the protocol
1046          * module may generate 3 segments over the wire:
1047          * 1460 bytes for the first, 448 (2048 - 1600) for the second, and
1048          * 892 for the third. If we "waste" 448 bytes in the first mblk,
1049          * the XDR encoding will generate two ~1400 byte mblks, and the
1050          * protocol module is more likely to produce properly sized segments.
1051          */
1052         if ((mpsize >> 1) <= len)
1053                 mp->b_rptr += (mpsize - len);

1055         /*
1056          * Adjust b_rptr to reserve space for the non-data protocol headers
1057          * any downstream modules might like to add, and for the
1058          * record marking header.
1059          */
1060         mp->b_rptr += (MSG_OFFSET + RM_HDR_SIZE);

1062         if (h->cl_auth->ah_cred.oa_flavor != RPCSEC_GSS) {
1063                 /* Copy in the preserialized RPC header information. */
1064                 bcopy(p->cku_rpchdr, mp->b_rptr, WIRE_HDR_SIZE);

1066                 /* Use XDR_SETPOS() to set the b_wptr to past the RPC header. */
1067                 XDR_SETPOS(xdrs, (uint_t)(mp->b_rptr - mp->b_datap->db_base +
1068                     WIRE_HDR_SIZE));

1070                 ASSERT((mp->b_wptr - mp->b_rptr) == WIRE_HDR_SIZE);

1072                 /* Serialize the procedure number and the arguments. */
1073                 if ((!XDR_PUTINT32(xdrs, (int32_t *)&procnum)) ||
1074                     (!AUTH_MARSHALL(h->cl_auth, xdrs, p->cku_cred)) ||
1075                     (!(*xdr_args)(xdrs, argsp))) {
1076                         p->cku_err.re_status = RPC_CANTENCODEARGS;
1077                         p->cku_err.re_errno = EIO;
1078                         goto cots_done;
1079                 }

1081                 (*(uint32_t *)(mp->b_rptr)) = p->cku_xid;
1082         } else {
1083                 uint32_t *uproc = (uint32_t *)&p->cku_rpchdr[WIRE_HDR_SIZE];
1084                 IXDR_PUT_U_INT32(uproc, procnum);

1086                 (*(uint32_t *)(&p->cku_rpchdr[0])) = p->cku_xid;
```



```
1088                 /* Use XDR_SETPOS() to set the b_wptr. */
1089                 XDR_SETPOS(xdrs, (uint_t)(mp->b_rptr - mp->b_datap->db_base));

1091                 /* Serialize the procedure number and the arguments. */
1092                 if (!AUTH_WRAP(h->cl_auth, p->cku_rpchdr, WIRE_HDR_SIZE+4,
1093                     xdrs, xdr_args, argsp)) {
1094                         p->cku_err.re_status = RPC_CANTENCODEARGS;
1095                         p->cku_err.re_errno = EIO;
1096                         goto cots_done;
1097                 }
1098         }

1100         RPCLOG(2, "clnt_cots_kcallit: connected, sending call, tidu_size %d\n",
1101             tidu_size);

1103         wq = cm_entry->x_wq;
1104         waitsecs = 0;

1106 dispatch_again:
1107         status = clnt_dispatch_send(wq, mp, call, p->cku_xid,
1108             (p->cku_flags & CKU_ONQUEUE));

1110         if ((status == RPC_CANTSEND) && (call->call_reason == ENOBUFS)) {
1111                 /*
1112                  * QFULL condition, allow some time for queue to drain
1113                  * and try again. Give up after waiting for all timeout
1114                  * specified for the call, or zone is going away.
1115                  */
1116                 max_waitsecs = wait.tv_sec ? wait.tv_sec : clnt_cots_min_tout;
1117                 if ((waitsecs++ < max_waitsecs) &&
1118                     !(zone_status_get(curproc->p_zone) >=
1119                     ZONE_IS_SHUTTING_DOWN)) {

1121                         /* wait 1 sec for queue to drain */
1122                         if (clnt_delay(drv_sectohz(1),
1122                         if (clnt_delay(drv_usectohz(1000000),
1123                             h->cl_nosignal) == EINTR) {
1124                                 p->cku_err.re_errno = EINTR;
1125                                 p->cku_err.re_status = RPC_INTR;

1127                                 goto cots_done;
1128                         }

1130                         /* and try again */
1131                         goto dispatch_again;
1132                 }
1133                 p->cku_err.re_status = status;
1134                 p->cku_err.re_errno = call->call_reason;
1135                 DTRACE_PROBE(krpc__e__clntcots__kcallit__cantsend);

1137                 goto cots_done;
1138         }

1140         if (waitsecs) {
1141                 /* adjust timeout to account for time wait to send */
1142                 wait.tv_sec -= waitsecs;
1143                 if (wait.tv_sec < 0) {
1144                         /* pick up reply on next retry */
1145                         wait.tv_sec = 0;
1146                 }
1147                 DTRACE_PROBE2(clnt_cots__sendwait, CLIENT *, h,
1148                     int, waitsecs);
1149         }

1151         RPCLOG(64, "clnt_cots_kcallit: sent call for xid 0x%x\n",
```

```
1152             (uint_t)p->cku_xid);
1153         p->cku_flags = (CKU_ONQUEUE|CKU_SENT);
1154         p->cku_recv_attempts = 1;

1156 #ifdef  RPCDEBUG
1157         time_sent = ddi_get_lbolt();
1158 #endif

1160         /*
1161          * Wait for a reply or a timeout.  If there is no error or timeout,
1162          * (both indicated by call_status), call->call_reply will contain
1163          * the RPC reply message.
1164          */
1165 read_again:
1166         mutex_enter(&call->call_lock);
1167         interrupted = 0;
1168         if (call->call_status == RPC_TIMEDOUT) {
1169                 /*
1170                  * Indicate that the lwp is not to be stopped while waiting
1171                  * for this network traffic.  This is to avoid deadlock while
1172                  * debugging a process via /proc and also to avoid recursive
1173                  * mutex_enter()s due to NFS page faults while stopping
1174                  * (NFS holds locks when it calls here).
1175                  */
1176                 clock_t cv_wait_ret;
1177                 clock_t timout;
1178                 clock_t oldlbolt;

1180                 klwp_t *lwp = ttolwp(curthread);

1182                 if (lwp != NULL)
1183                         lwp->lwp_nostop++;

1185                 oldlbolt = ddi_get_lbolt();
1186                 timout = drv_sectohz(wait.tv_sec) +
1186                 timout = wait.tv_sec * drv_usectohz(1000000) +
1187                     drv_usectohz(wait.tv_usec) + oldlbolt;
1188                 /*
1189                  * Iterate until the call_status is changed to something
1190                  * other that RPC_TIMEDOUT, or if cv_timedwait_sig() returns
1191                  * something <=0 zero. The latter means that we timed
1192                  * out.
1193                  */
1194                 if (h->cl_nosignal)
1195                         while ((cv_wait_ret = cv_timedwait(&call->call_cv,
1196                             &call->call_lock, timout)) > 0 &&
1197                             call->call_status == RPC_TIMEDOUT)
1198                                 ;
1199                 else
1200                         while ((cv_wait_ret = cv_timedwait_sig(
1201                             &call->call_cv,
1202                             &call->call_lock, timout)) > 0 &&
1203                             call->call_status == RPC_TIMEDOUT)
1204                                 ;

1206                 switch (cv_wait_ret) {
1207                 case 0:
1208                         /*
1209                          * If we got out of the above loop with
1210                          * cv_timedwait_sig() returning 0, then we were
1211                          * interrupted regardless what call_status is.
1212                          */
1213                         interrupted = 1;
1214                         break;
1215                 case -1:
1216                         /* cv_timedwait_sig() timed out */
```



```
1217                         break;
1218                 default:

1220                         /*
1221                          * We were cv_signaled(). If we didn't
1222                          * get a successful call_status and returned
1223                          * before time expired, delay up to clnt_cots_min_tout
1224                          * seconds so that the caller doesn't immediately
1225                          * try to call us again and thus force the
1226                          * same condition that got us here (such
1227                          * as a RPC_XPRTFAILED due to the server not
1228                          * listening on the end-point.
1229                          */
1230                         if (call->call_status != RPC_SUCCESS) {
1231                                 clock_t curlbolt;
1232                                 clock_t diff;

1234                                 curlbolt = ddi_get_lbolt();
1235                                 ticks = drv_sectohz(clnt_cots_min_tout);
1235                                 ticks = clnt_cots_min_tout *
1236                                     drv_usectohz(1000000);
1236                                 diff = curlbolt - oldlbolt;
1237                                 if (diff < ticks) {
1238                                         delay_first = TRUE;
1239                                         if (diff > 0)
1240                                                 ticks -= diff;
1241                                 }
1242                         }
1243                         break;
1244                 }

1246                 if (lwp != NULL)
1247                         lwp->lwp_nostop--;
1248         }
1249         /*
1250          * Get the reply message, if any.  This will be freed at the end
1251          * whether or not an error occurred.
1252          */
1253         mp = call->call_reply;
1254         call->call_reply = NULL;

1256         /*
1257          * call_err is the error info when the call is on dispatch queue.
1258          * cku_err is the error info returned to the caller.
1259          * Sync cku_err with call_err for local message processing.
1260          */

1262         status = call->call_status;
1263         p->cku_err = call->call_err;
1264         mutex_exit(&call->call_lock);

1266         if (status != RPC_SUCCESS) {
1267                 switch (status) {
1268                 case RPC_TIMEDOUT:
1269                         now = ddi_get_lbolt();
1270                         if (interrupted) {
1271                                 COTSRCSTAT_INCR(p->cku_stats, rcintrs);
1272                                 p->cku_err.re_status = RPC_INTR;
1273                                 p->cku_err.re_errno = EINTR;
1274                                 RPCLOG(1, "clnt_cots_kcallit: xid 0x%x",
1275                                     p->cku_xid);
1276                                 RPCLOG(1, "signal interrupted at %ld", now);
1277                                 RPCLOG(1, ", was sent at %ld\n", time_sent);
1278                         } else {
1279                                 COTSRCSTAT_INCR(p->cku_stats, rctimeouts);
1280                                 p->cku_err.re_errno = ETIMEDOUT;
```

```
1281                                RPCLOG(1, "clnt_cots_kcallit: timed out at %ld",
1282                                    now);
1283                                RPCLOG(1, ", was sent at %ld\n", time_sent);
1284                        }
1285                        break;

1287                case RPC_XPRTFAILED:
1288                        if (p->cku_err.re_errno == 0)
1289                                p->cku_err.re_errno = EIO;

1291                        RPCLOG(1, "clnt_cots_kcallit: transport failed: %d\n",
1292                            p->cku_err.re_errno);
1293                        break;

1295                case RPC_SYSTEMERROR:
1296                        ASSERT(p->cku_err.re_errno);
1297                        RPCLOG(1, "clnt_cots_kcallit: system error: %d\n",
1298                            p->cku_err.re_errno);
1299                        break;

1301                default:
1302                        p->cku_err.re_status = RPC_SYSTEMERROR;
1303                        p->cku_err.re_errno = EIO;
1304                        RPCLOG(1, "clnt_cots_kcallit: error: %s\n",
1305                            clnt_sperrno(status));
1306                        break;
1307                }
1308                if (p->cku_err.re_status != RPC_TIMEDOUT) {

1310                        if (p->cku_flags & CKU_ONQUEUE) {
1311                                call_table_remove(call);
1312                                p->cku_flags &= ~CKU_ONQUEUE;
1313                        }

1315                        RPCLOG(64, "clnt_cots_kcallit: non TIMEOUT so xid 0x%x "
1316                            "taken off dispatch list\n", p->cku_xid);
1317                        if (call->call_reply) {
1318                                freemsg(call->call_reply);
1319                                call->call_reply = NULL;
1320                        }
1321                } else if (wait.tv_sec != 0) {
1322                        /*
1323                         * We've sent the request over TCP and so we have
1324                         * every reason to believe it will get
1325                         * delivered. In which case returning a timeout is not
1326                         * appropriate.
1327                         */
1328                        if (p->cku_progress == TRUE &&
1329                            p->cku_recv_attempts < clnt_cots_maxrecv) {
1330                                p->cku_err.re_status = RPC_INPROGRESS;
1331                        }
1332                }
1333                goto cots_done;
1334        }

1336        xdrs = &p->cku_inxdr;
1337        xdrmblk_init(xdrs, mp, XDR_DECODE, 0);

1339        reply_msg.rm_direction = REPLY;
1340        reply_msg.rm_reply.rp_stat = MSG_ACCEPTED;
1341        reply_msg.acpted_rply.ar_stat = SUCCESS;

1343        reply_msg.acpted_rply.ar_verf = _null_auth;
1344        /*
1345         *  xdr_results will be done in AUTH_UNWRAP.
1346         */
```



```
1347        reply_msg.acpted_rply.ar_results.where = NULL;
1348        reply_msg.acpted_rply.ar_results.proc = xdr_void;

1350        if (xdr_replymsg(xdrs, &reply_msg)) {
1351                enum clnt_stat re_status;

1353                _seterr_reply(&reply_msg, &p->cku_err);

1355                re_status = p->cku_err.re_status;
1356                if (re_status == RPC_SUCCESS) {
1357                        /*
1358                         * Reply is good, check auth.
1359                         */
1360                        if (!AUTH_VALIDATE(h->cl_auth,
1361                            &reply_msg.acpted_rply.ar_verf)) {
1362                                COTSRCSTAT_INCR(p->cku_stats, rcbadverfs);
1363                                RPCLOG0(1, "clnt_cots_kcallit: validation "
1364                                    "failure\n");
1365                                freemsg(mp);
1366                                (void) xdr_rpc_free_verifier(xdrs, &reply_msg);
1367                                mutex_enter(&call->call_lock);
1368                                if (call->call_reply == NULL)
1369                                        call->call_status = RPC_TIMEDOUT;
1370                                mutex_exit(&call->call_lock);
1371                                goto read_again;
1372                        } else if (!AUTH_UNWRAP(h->cl_auth, xdrs,
1373                            xdr_results, resultsp)) {
1374                                RPCLOG0(1, "clnt_cots_kcallit: validation "
1375                                    "failure (unwrap)\n");
1376                                p->cku_err.re_status = RPC_CANTDECODERES;
1377                                p->cku_err.re_errno = EIO;
1378                        }
1379                } else {
1380                        /* set errno in case we can't recover */
1381                        if (re_status != RPC_VERSMISMATCH &&
1382                            re_status != RPC_AUTHERROR &&
1383                            re_status != RPC_PROGVERSMISMATCH)
1384                                p->cku_err.re_errno = EIO;

1386                        if (re_status == RPC_AUTHERROR) {
1387                                /*
1388                                 * Maybe our credential need to be refreshed
1389                                 */
1390                                if (cm_entry) {
1391                                        /*
1392                                         * There is the potential that the
1393                                         * cm_entry has/will be marked dead,
1394                                         * so drop the connection altogether,
1395                                         * force REFRESH to establish new
1396                                         * connection.
1397                                         */
1398                                        connmgr_cancelconn(cm_entry);
1399                                        cm_entry = NULL;
1400                                }

1402                                (void) xdr_rpc_free_verifier(xdrs,
1403                                    &reply_msg);

1405                                if (p->cku_flags & CKU_ONQUEUE) {
1406                                        call_table_remove(call);
1407                                        p->cku_flags &= ~CKU_ONQUEUE;
1408                                }
1409                                RPCLOG(64,
1410                                    "clnt_cots_kcallit: AUTH_ERROR, xid"
1411                                    " 0x%x removed off dispatch list\n",
1412                                    p->cku_xid);
```

```
1413                            if (call->call_reply) {
1414                                    freemsg(call->call_reply);
1415                                    call->call_reply = NULL;
1416                            }

1418                            if ((refreshes > 0) &&
1419                                AUTH_REFRESH(h->cl_auth, &reply_msg,
1420                                p->cku_cred)) {
1421                                    refreshes--;
1422                                    freemsg(mp);
1423                                    mp = NULL;

1425                                    COTSRCSTAT_INCR(p->cku_stats,
1426                                        rcbadcalls);
1427                                    COTSRCSTAT_INCR(p->cku_stats,
1428                                        rcnewcreds);
1429                                    goto call_again;
1430                            }

1432                            /*
1433                             * We have used the client handle to
1434                             * do an AUTH_REFRESH and the RPC status may
1435                             * be set to RPC_SUCCESS; Let's make sure to
1436                             * set it to RPC_AUTHERROR.
1437                             */
1438                            p->cku_err.re_status = RPC_AUTHERROR;

1440                            /*
1441                             * Map recoverable and unrecoverable
1442                             * authentication errors to appropriate errno
1443                             */
1444                            switch (p->cku_err.re_why) {
1445                            case AUTH_TOOWEAK:
1446                                    /*
1447                                     * This could be a failure where the
1448                                     * server requires use of a reserved
1449                                     * port,  check and optionally set the
1450                                     * client handle useresvport trying
1451                                     * one more time. Next go round we
1452                                     * fall out with the tooweak error.
1453                                     */
1454                                    if (p->cku_useresvport != 1) {
1455                                            p->cku_useresvport = 1;
1456                                            p->cku_xid = 0;
1457                                            freemsg(mp);
1458                                            mp = NULL;
1459                                            goto call_again;
1460                                    }
1461                                    /* FALLTHRU */
1462                            case AUTH_BADCRED:
1463                            case AUTH_BADVERF:
1464                            case AUTH_INVALIDRESP:
1465                            case AUTH_FAILED:
1466                            case RPCSEC_GSS_NOCRED:
1467                            case RPCSEC_GSS_FAILED:
1468                                    p->cku_err.re_errno = EACCES;
1469                                    break;
1470                            case AUTH_REJECTEDCRED:
1471                            case AUTH_REJECTEDVERF:
1472                            default:        p->cku_err.re_errno = EIO;
1473                                    break;
1474                            }
1475                            RPCLOG(1, "clnt_cots_kcallit : authentication"
1476                                " failed with RPC_AUTHERROR of type %d\n",
1477                                (int)p->cku_err.re_why);
1478                            goto cots_done;
```



```
1479                                    }
1480                            }
1481                    } else {
1482                            /* reply didn't decode properly. */
1483                            p->cku_err.re_status = RPC_CANTDECODERES;
1484                            p->cku_err.re_errno = EIO;
1485                            RPCLOG0(1, "clnt_cots_kcallit: decode failure\n");
1486                    }

1488                    (void) xdr_rpc_free_verifier(xdrs, &reply_msg);

1490                    if (p->cku_flags & CKU_ONQUEUE) {
1491                            call_table_remove(call);
1492                            p->cku_flags &= ~CKU_ONQUEUE;
1493                    }

1495                    RPCLOG(64, "clnt_cots_kcallit: xid 0x%x taken off dispatch list",
1496                        p->cku_xid);
1497                    RPCLOG(64, " status is %s\n", clnt_sperrno(p->cku_err.re_status));
1498            cots_done:
1499                    if (cm_entry)
1500                            connmgr_release(cm_entry);

1502                    if (mp != NULL)
1503                            freemsg(mp);
1504                    if ((p->cku_flags & CKU_ONQUEUE) == 0 && call->call_reply) {
1505                            freemsg(call->call_reply);
1506                            call->call_reply = NULL;
1507                    }
1508                    if (p->cku_err.re_status != RPC_SUCCESS) {
1509                            RPCLOG0(1, "clnt_cots_kcallit: tail-end failure\n");
1510                            COTSRCSTAT_INCR(p->cku_stats, rcbadcalls);
1511                    }

1513                    /*
1514                     * No point in delaying if the zone is going away.
1515                     */
1516                    if (delay_first == TRUE &&
1517                        !(zone_status_get(curproc->p_zone) >= ZONE_IS_SHUTTING_DOWN)) {
1518                            if (clnt_delay(ticks, h->cl_nosignal) == EINTR) {
1519                                    p->cku_err.re_errno = EINTR;
1520                                    p->cku_err.re_status = RPC_INTR;
1521                            }
1522                    }
1523                    return (p->cku_err.re_status);
1524            }
______unchanged_portion_omitted_

1688 /*
1689  * Wait for a connection until a timeout, or until we are
1690  * signalled that there has been a connection state change.
1691  */
1692 static enum clnt_stat
1693 connmgr_cwait(struct cm_xprt *cm_entry, const struct timeval *waitp,
1694         bool_t nosignal)
1695 {
1696         bool_t interrupted;
1697         clock_t timout, cv_stat;
1698         enum clnt_stat clstat;
1699         unsigned int old_state;

1701         ASSERT(MUTEX_HELD(&connmgr_lock));
1702         /*
1703          * We wait for the transport connection to be made, or an
1704          * indication that it could not be made.
1705          */
```

```
1706         clstat = RPC_TIMEDOUT;
1707         interrupted = FALSE;

1709         old_state = cm_entry->x_state_flags;
1710         /*
1711          * Now loop until cv_timedwait{_sig} returns because of
1712          * a signal(0) or timeout(-1) or cv_signal(>0). But it may be
1713          * cv_signalled for various other reasons too. So loop
1714          * until there is a state change on the connection.
1715          */

1717         timout = drv_sectohz(waitp->tv_sec) +
1718         timout = waitp->tv_sec * drv_usectohz(1000000) +
1718             drv_usectohz(waitp->tv_usec) + ddi_get_lbolt();

1720         if (nosignal) {
1721                 while ((cv_stat = cv_timedwait(&cm_entry->x_conn_cv,
1722                     &connmgr_lock, timout)) > 0 &&
1723                     cm_entry->x_state_flags == old_state)
1724                         ;
1725         } else {
1726                 while ((cv_stat = cv_timedwait_sig(&cm_entry->x_conn_cv,
1727                     &connmgr_lock, timout)) > 0 &&
1728                     cm_entry->x_state_flags == old_state)
1729                         ;

1731                 if (cv_stat == 0) /* got intr signal? */
1732                         interrupted = TRUE;
1733         }

1735         if ((cm_entry->x_state_flags & (X_BADSTATES|X_CONNECTED)) ==
1736             X_CONNECTED) {
1737                 clstat = RPC_SUCCESS;
1738         } else {
1739                 if (interrupted == TRUE)
1740                         clstat = RPC_INTR;
1741                 RPCLOG(1, "connmgr_cwait: can't connect, error: %s\n",
1742                     clnt_sperrno(clstat));
1743         }

1745         return (clstat);
1746 }
______unchanged_portion_omitted_

2303 /*
2304  * Keep the cm_xprt entry on the connecton list when making a connection. This
2305  * is to prevent multiple connections to a slow server from appearing.
2306  * We use the bit field x_thread to tell if a thread is doing a connection
2307  * which keeps other interested threads from messing with connection.
2308  * Those other threads just wait if x_thread is set.
2309  *
2310  * If x_thread is not set, then we do the actual work of connecting via
2311  * connmgr_connect().
2312  *
2313  * mutex convention: called with connmgr_lock held, returns with it released.
2314  */
2315 static struct cm_xprt *
2316 connmgr_wrapconnect(
2317         struct cm_xprt  *cm_entry,
2318         const struct timeval    *waitp,
2319         struct netbuf   *destaddr,
2320         int             addrfmly,
2321         struct netbuf   *srcaddr,
2322         struct rpc_err  *rpcerr,
2323         bool_t          reconnect,
2324         bool_t          nosignal,
```



```
2325         cred_t          *cr)
2326 {
2327         ASSERT(MUTEX_HELD(&connmgr_lock));
2328         /*
2329          * Hold this entry as we are about to drop connmgr_lock.
2330          */
2331         CONN_HOLD(cm_entry);

2333         /*
2334          * If there is a thread already making a connection for us, then
2335          * wait for it to complete the connection.
2336          */
2337         if (cm_entry->x_thread == TRUE) {
2338                 rpcerr->re_status = connmgr_cwait(cm_entry, waitp, nosignal);

2340                 if (rpcerr->re_status != RPC_SUCCESS) {
2341                         mutex_exit(&connmgr_lock);
2342                         connmgr_release(cm_entry);
2343                         return (NULL);
2344                 }
2345         } else {
2346                 bool_t connected;
2347                 calllist_t call;

2349                 cm_entry->x_thread = TRUE;

2351                 while (cm_entry->x_needrel == TRUE) {
2352                         cm_entry->x_needrel = FALSE;

2354                         connmgr_sndrel(cm_entry);
2355                         delay(drv_sectohz(1));
2356                         delay(drv_usectohz(1000000));

2357                         mutex_enter(&connmgr_lock);
2358                 }

2360                 /*
2361                  * If we need to send a T_DISCON_REQ, send one.
2362                  */
2363                 connmgr_dis_and_wait(cm_entry);

2365                 mutex_exit(&connmgr_lock);

2367                 bzero(&call, sizeof (call));
2368                 cv_init(&call.call_cv, NULL, CV_DEFAULT, NULL);

2370                 connected = connmgr_connect(cm_entry, cm_entry->x_wq,
2371                     destaddr, addrfmly, &call, &cm_entry->x_tidu_size,
2372                     reconnect, waitp, nosignal, cr);

2374                 *rpcerr = call.call_err;
2375                 cv_destroy(&call.call_cv);

2377                 mutex_enter(&connmgr_lock);


2380                 if (cm_entry->x_early_disc) {
2381                         /*
2382                          * We need to check if a disconnect request has come
2383                          * while we are connected, if so, then we need to
2384                          * set rpcerr->re_status appropriately before returning
2385                          * NULL to caller.
2386                          */
2387                         if (rpcerr->re_status == RPC_SUCCESS)
2388                                 rpcerr->re_status = RPC_XPRTFAILED;
2389                         cm_entry->x_connected = FALSE;
```

```
2390                } else
2391                        cm_entry->x_connected = connected;

2393                /*
2394                 * There could be a discrepancy here such that
2395                 * x_early_disc is TRUE yet connected is TRUE as well
2396                 * and the connection is actually connected. In that case
2397                 * lets be conservative and declare the connection as not
2398                 * connected.
2399                 */

2401                cm_entry->x_early_disc = FALSE;
2402                cm_entry->x_needdis = (cm_entry->x_connected == FALSE);


2405                /*
2406                 * connmgr_connect() may have given up before the connection
2407                 * actually timed out. So ensure that before the next
2408                 * connection attempt we do a disconnect.
2409                 */
2410                cm_entry->x_ctime = ddi_get_lbolt();
2411                cm_entry->x_thread = FALSE;

2413                cv_broadcast(&cm_entry->x_conn_cv);

2415                if (cm_entry->x_connected == FALSE) {
2416                        mutex_exit(&connmgr_lock);
2417                        connmgr_release(cm_entry);
2418                        return (NULL);
2419                }
2420        }

2422        if (srcaddr != NULL) {
2423                /*
2424                 * Copy into the handle the
2425                 * source address of the
2426                 * connection, which we will use
2427                 * in case of a later retry.
2428                 */
2429                if (srcaddr->len != cm_entry->x_src.len) {
2430                        if (srcaddr->maxlen > 0)
2431                                kmem_free(srcaddr->buf, srcaddr->maxlen);
2432                        srcaddr->buf = kmem_zalloc(cm_entry->x_src.len,
2433                            KM_SLEEP);
2434                        srcaddr->maxlen = srcaddr->len =
2435                            cm_entry->x_src.len;
2436                }
2437                bcopy(cm_entry->x_src.buf, srcaddr->buf, srcaddr->len);
2438        }
2439        cm_entry->x_time = ddi_get_lbolt();
2440        mutex_exit(&connmgr_lock);
2441        return (cm_entry);
2442 }

2444 /*
2445  * If we need to send a T_DISCON_REQ, send one.
2446  */
2447 static void
2448 connmgr_dis_and_wait(struct cm_xprt *cm_entry)
2449 {
2450        ASSERT(MUTEX_HELD(&connmgr_lock));
2451        for (;;) {
2452                while (cm_entry->x_needdis == TRUE) {
2453                        RPCLOG(8, "connmgr_dis_and_wait: need "
2454                            "T_DISCON_REQ for connection 0x%p\n",
2455                            (void *)cm_entry);
```



```
2456                        cm_entry->x_needdis = FALSE;
2457                        cm_entry->x_waitdis = TRUE;

2459                        connmgr_snddis(cm_entry);

2461                        mutex_enter(&connmgr_lock);
2462                }

2464                if (cm_entry->x_waitdis == TRUE) {
2465                        clock_t timout;

2467                        RPCLOG(8, "connmgr_dis_and_wait waiting for "
2468                            "T_DISCON_REQ's ACK for connection %p\n",
2469                            (void *)cm_entry);

2471                        timout = drv_sectohz(clnt_cots_min_conntout);
2472                        timout = clnt_cots_min_conntout * drv_usectohz(1000000);

2473                        /*
2474                         * The TPI spec says that the T_DISCON_REQ
2475                         * will get acknowledged, but in practice
2476                         * the ACK may never get sent. So don't
2477                         * block forever.
2478                         */
2479                        (void) cv_reltimedwait(&cm_entry->x_dis_cv,
2480                            &connmgr_lock, timout, TR_CLOCK_TICK);
2481                }
2482                /*
2483                 * If we got the ACK, break. If we didn't,
2484                 * then send another T_DISCON_REQ.
2485                 */
2486                if (cm_entry->x_waitdis == FALSE) {
2487                        break;
2488                } else {
2489                        RPCLOG(8, "connmgr_dis_and_wait: did"
2490                            "not get T_DISCON_REQ's ACK for "
2491                            "connection  %p\n", (void *)cm_entry);
2492                        cm_entry->x_needdis = TRUE;
2493                }
2494        }
2495 }
______unchanged_portion_omitted_

3694 /*
3695  * Wait for TPI ack, returns success only if expected ack is received
3696  * within timeout period.
3697  */

3699 static int
3700 waitforack(calllist_t *e, t_scalar_t ack_prim, const struct timeval *waitp,
3701     bool_t nosignal)
3702 {
3703        union T_primitives *tpr;
3704        clock_t timout;
3705        int cv_stat = 1;

3707        ASSERT(MUTEX_HELD(&clnt_pending_lock));
3708        while (e->call_reply == NULL) {
3709                if (waitp != NULL) {
3710                        timout = drv_sectohz(waitp->tv_sec) +
3711                        timout = waitp->tv_sec * drv_usectohz(MICROSEC) +
3711                            drv_usectohz(waitp->tv_usec);
3712                        if (nosignal)
3713                                cv_stat = cv_reltimedwait(&e->call_cv,
3714                                    &clnt_pending_lock, timout, TR_CLOCK_TICK);
3715                        else
```

```
3716                                    cv_stat = cv_reltimedwait_sig(&e->call_cv,
3717                                        &clnt_pending_lock, timout, TR_CLOCK_TICK);
3718                            } else {
3719                                    if (nosignal)
3720                                            cv_wait(&e->call_cv, &clnt_pending_lock);
3721                                    else
3722                                            cv_stat = cv_wait_sig(&e->call_cv,
3723                                                &clnt_pending_lock);
3724                            }
3725                            if (cv_stat == -1)
3726                                    return (ETIME);
3727                            if (cv_stat == 0)
3728                                    return (EINTR);
3729                            /*
3730                             * if we received an error from the server and we know a reply
3731                             * is not going to be sent, do not wait for the full timeout,
3732                             * return now.
3733                             */
3734                            if (e->call_status == RPC_XPRTFAILED)
3735                                    return (e->call_reason);
3736                    }
3737                    tpr = (union T_primitives *)e->call_reply->b_rptr;
3738                    if (tpr->type == ack_prim)
3739                            return (0); /* Success */

3741                    if (tpr->type == T_ERROR_ACK) {
3742                            if (tpr->error_ack.TLI_error == TSYSERR)
3743                                    return (tpr->error_ack.UNIX_error);
3744                            else
3745                                    return (t_tlitosyserr(tpr->error_ack.TLI_error));
3746                    }

3748                    return (EPROTO); /* unknown or unexpected primitive */
3749 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35269 Wed Aug 19 07:25:22 2015
new/usr/src/uts/common/rpc/clnt_rdma.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 561 /* ARGSUSED */
 562 static enum clnt_stat
 563 clnt_rdma_kcallit(CLIENT *h, rpcproc_t procnum, xdrproc_t xdr_args,
 564     caddr_t argsp, xdrproc_t xdr_results, caddr_t resultsp,
 565     struct timeval wait)
 566 {
 567         cku_private_t *p = htop(h);

 569         int     try_call_again;
 570         int     refresh_attempt = AUTH_REFRESH_COUNT;
 571         int     status;
 572         int     msglen;

 574         XDR     *call_xdrp, callxdr; /* for xdrrdma encoding the RPC call */
 575         XDR     *reply_xdrp, replyxdr; /* for xdrrdma decoding the RPC reply */
 576         XDR     *rdmahdr_o_xdrs, *rdmahdr_i_xdrs;

 578         struct rpc_msg  reply_msg;
 579         rdma_registry_t *m;

 581         struct clist *cl_sendlist;
 582         struct clist *cl_recvlist;
 583         struct clist *cl;
 584         struct clist *cl_rpcmsg;
 585         struct clist *cl_rdma_reply;
 586         struct clist *cl_rpcreply_wlist;
 587         struct clist *cl_long_reply;
 588         rdma_buf_t  rndup;

 590         uint_t vers;
 591         uint_t op;
 592         uint_t off;
 593         uint32_t seg_array_len;
 594         uint_t long_reply_len;
 595         uint_t rpcsec_gss;
 596         uint_t gss_i_or_p;

 598         CONN *conn = NULL;
 599         rdma_buf_t clmsg;
 600         rdma_buf_t rpcmsg;
 601         rdma_chunkinfo_lengths_t rcil;

 603         clock_t ticks;
 604         bool_t wlist_exists_reply;

 606         uint32_t rdma_credit = rdma_bufs_rqst;

 608         RCSTAT_INCR(rccalls);

 610 call_again:

 612         bzero(&clmsg, sizeof (clmsg));
 613         bzero(&rpcmsg, sizeof (rpcmsg));
 614         bzero(&rndup, sizeof (rndup));
 615         try_call_again = 0;
 616         cl_sendlist = NULL;
 617         cl_recvlist = NULL;
 618         cl = NULL;
 619         cl_rpcmsg = NULL;
```



```
 620         cl_rdma_reply = NULL;
 621         call_xdrp = NULL;
 622         reply_xdrp = NULL;
 623         wlist_exists_reply  = FALSE;
 624         cl_rpcreply_wlist = NULL;
 625         cl_long_reply = NULL;
 626         rcil.rcil_len = 0;
 627         rcil.rcil_len_alt = 0;
 628         long_reply_len = 0;

 630         rw_enter(&rdma_lock, RW_READER);
 631         m = (rdma_registry_t *)p->cku_rd_handle;
 632         if (m->r_mod_state == RDMA_MOD_INACTIVE) {
 633                 /*
 634                  * If we didn't find a matching RDMA module in the registry
 635                  * then there is no transport.
 636                  */
 637                 rw_exit(&rdma_lock);
 638                 p->cku_err.re_status = RPC_CANTSEND;
 639                 p->cku_err.re_errno = EIO;
 640                 ticks = drv_sectohz(clnt_rdma_min_delay);
 640                 ticks = clnt_rdma_min_delay * drv_usectohz(1000000);
 641                 if (h->cl_nosignal == TRUE) {
 642                         delay(ticks);
 643                 } else {
 644                         if (delay_sig(ticks) == EINTR) {
 645                                 p->cku_err.re_status = RPC_INTR;
 646                                 p->cku_err.re_errno = EINTR;
 647                         }
 648                 }
 649                 return (RPC_CANTSEND);
 650         }
 651         /*
 652          * Get unique xid
 653          */
 654         if (p->cku_xid == 0)
 655                 p->cku_xid = alloc_xid();

 657         status = RDMA_GET_CONN(p->cku_rd_mod->rdma_ops, &p->cku_srcaddr,
 658             &p->cku_addr, p->cku_addrfmly, p->cku_rd_handle, &conn);
 659         rw_exit(&rdma_lock);

 661         /*
 662          * If there is a problem with the connection reflect the issue
 663          * back to the higher level to address, we MAY delay for a short
 664          * period so that we are kind to the transport.
 665          */
 666         if (conn == NULL) {
 667                 /*
 668                  * Connect failed to server. Could be because of one
 669                  * of several things. In some cases we don't want
 670                  * the caller to retry immediately - delay before
 671                  * returning to caller.
 672                  */
 673                 switch (status) {
 674                 case RDMA_TIMEDOUT:
 675                         /*
 676                          * Already timed out. No need to delay
 677                          * some more.
 678                          */
 679                         p->cku_err.re_status = RPC_TIMEDOUT;
 680                         p->cku_err.re_errno = ETIMEDOUT;
 681                         break;
 682                 case RDMA_INTR:
 683                         /*
 684                          * Failed because of an signal. Very likely
```

```
685                          * the caller will not retry.
686                          */
687                         p->cku_err.re_status = RPC_INTR;
688                         p->cku_err.re_errno = EINTR;
689                         break;
690                 default:
691                         /*
692                          * All other failures - server down or service
693                          * down or temporary resource failure. Delay before
694                          * returning to caller.
695                          */
696                         ticks = drv_sectohz(clnt_rdma_min_delay);
696                         ticks = clnt_rdma_min_delay * drv_usectohz(1000000);
697                         p->cku_err.re_status = RPC_CANTCONNECT;
698                         p->cku_err.re_errno = EIO;

700                         if (h->cl_nosignal == TRUE) {
701                                 delay(ticks);
702                         } else {
703                                 if (delay_sig(ticks) == EINTR) {
704                                         p->cku_err.re_status = RPC_INTR;
705                                         p->cku_err.re_errno = EINTR;
706                                 }
707                         }
708                         break;
709                 }

711                 return (p->cku_err.re_status);
712         }

714         if (p->cku_srcaddr.maxlen < conn->c_laddr.len) {
715                 if ((p->cku_srcaddr.maxlen != 0) &&
716                     (p->cku_srcaddr.buf != NULL))
717                         kmem_free(p->cku_srcaddr.buf, p->cku_srcaddr.maxlen);
718                 p->cku_srcaddr.buf = kmem_zalloc(conn->c_laddr.maxlen,
719                     KM_SLEEP);
720                 p->cku_srcaddr.maxlen = conn->c_laddr.maxlen;
721         }

723         p->cku_srcaddr.len = conn->c_laddr.len;
724         bcopy(conn->c_laddr.buf, p->cku_srcaddr.buf, conn->c_laddr.len);

726         clnt_check_credit(conn);

728         status = CLNT_RDMA_FAIL;

730         rpcsec_gss = gss_i_or_p = FALSE;

732         if (IS_RPCSEC_GSS(h)) {
733                 rpcsec_gss = TRUE;
734                 if (rpc_gss_get_service_type(h->cl_auth) ==
735                     rpc_gss_svc_integrity ||
736                     rpc_gss_get_service_type(h->cl_auth) ==
737                     rpc_gss_svc_privacy)
738                         gss_i_or_p = TRUE;
739         }

741         /*
742          * Try a regular RDMA message if RPCSEC_GSS is not being used
743          * or if RPCSEC_GSS is being used for authentication only.
744          */
745         if (rpcsec_gss == FALSE ||
746             (rpcsec_gss == TRUE && gss_i_or_p == FALSE)) {
747                 /*
748                  * Grab a send buffer for the request.  Try to
749                  * encode it to see if it fits. If not, then it
```



```
750                  * needs to be sent in a chunk.
751                  */
752                 rpcmsg.type = SEND_BUFFER;
753                 if (rdma_buf_alloc(conn, &rpcmsg)) {
754                         DTRACE_PROBE(krpc__e__clntrdma__callit_nobufs);
755                         goto done;
756                 }

758                 /* First try to encode into regular send buffer */
759                 op = RDMA_MSG;

761                 call_xdrp = &callxdr;

763                 xdrrdma_create(call_xdrp, rpcmsg.addr, rpcmsg.len,
764                     rdma_minchunk, NULL, XDR_ENCODE, conn);

766                 status = clnt_compose_rpcmsg(h, procnum, &rpcmsg, call_xdrp,
767                     xdr_args, argsp);

769                 if (status != CLNT_RDMA_SUCCESS) {
770                         /* Clean up from previous encode attempt */
771                         rdma_buf_free(conn, &rpcmsg);
772                         XDR_DESTROY(call_xdrp);
773                 } else {
774                         XDR_CONTROL(call_xdrp, XDR_RDMA_GET_CHUNK_LEN, &rcil);
775                 }
776         }

778         /* If the encode didn't work, then try a NOMSG */
779         if (status != CLNT_RDMA_SUCCESS) {

781                 msglen = CKU_HDRSIZE + BYTES_PER_XDR_UNIT + MAX_AUTH_BYTES +
782                     xdr_sizeof(xdr_args, argsp);

784                 msglen = calc_length(msglen);

786                 /* pick up the lengths for the reply buffer needed */
787                 (void) xdrrdma_sizeof(xdr_args, argsp, 0,
788                     &rcil.rcil_len, &rcil.rcil_len_alt);

790                 /*
791                  * Construct a clist to describe the CHUNK_BUFFER
792                  * for the rpcmsg.
793                  */
794                 cl_rpcmsg = clist_alloc();
795                 cl_rpcmsg->c_len = msglen;
796                 cl_rpcmsg->rb_longbuf.type = RDMA_LONG_BUFFER;
797                 cl_rpcmsg->rb_longbuf.len = msglen;
798                 if (rdma_buf_alloc(conn, &cl_rpcmsg->rb_longbuf)) {
799                         clist_free(cl_rpcmsg);
800                         goto done;
801                 }
802                 cl_rpcmsg->w.c_saddr3 = cl_rpcmsg->rb_longbuf.addr;

804                 op = RDMA_NOMSG;
805                 call_xdrp = &callxdr;

807                 xdrrdma_create(call_xdrp, cl_rpcmsg->rb_longbuf.addr,
808                     cl_rpcmsg->rb_longbuf.len, 0,
809                     cl_rpcmsg, XDR_ENCODE, conn);

811                 status = clnt_compose_rpcmsg(h, procnum, &cl_rpcmsg->rb_longbuf,
812                     call_xdrp, xdr_args, argsp);

814                 DTRACE_PROBE2(krpc__i__clntrdma__callit__longbuf, int, status,
815                     int, msglen);
```

```
816                 if (status != CLNT_RDMA_SUCCESS) {
817                         p->cku_err.re_status = RPC_CANTENCODEARGS;
818                         p->cku_err.re_errno = EIO;
819                         DTRACE_PROBE(krpc__e__clntrdma__callit__composemsg);
820                         goto done;
821                 }
822         }

824         /*
825          * During the XDR_ENCODE we may have "allocated" an RDMA READ or
826          * RDMA WRITE clist.
827          *
828          * First pull the RDMA READ chunk list from the XDR private
829          * area to keep it handy.
830          */
831         XDR_CONTROL(call_xdrp, XDR_RDMA_GET_RLIST, &cl);

833         if (gss_i_or_p) {
834                 long_reply_len = rcil.rcil_len + rcil.rcil_len_alt;
835                 long_reply_len += MAX_AUTH_BYTES;
836         } else {
837                 long_reply_len = rcil.rcil_len;
838         }

840         /*
841          * Update the chunk size information for the Long RPC msg.
842          */
843         if (cl && op == RDMA_NOMSG)
844                 cl->c_len = p->cku_outsz;

846         /*
847          * Prepare the RDMA header. On success xdrs will hold the result
848          * of xdrmem_create() for a SEND_BUFFER.
849          */
850         status = clnt_compose_rdma_header(conn, h, &clmsg,
851             &rdmahdr_o_xdrs, &op);

853         if (status != CLNT_RDMA_SUCCESS) {
854                 p->cku_err.re_status = RPC_CANTSEND;
855                 p->cku_err.re_errno = EIO;
856                 RCSTAT_INCR(rcnomem);
857                 DTRACE_PROBE(krpc__e__clntrdma__callit__nobufs2);
858                 goto done;
859         }

861         /*
862          * Now insert the RDMA READ list iff present
863          */
864         status = clnt_setup_rlist(conn, rdmahdr_o_xdrs, call_xdrp);
865         if (status != CLNT_RDMA_SUCCESS) {
866                 DTRACE_PROBE(krpc__e__clntrdma__callit__clistreg);
867                 rdma_buf_free(conn, &clmsg);
868                 p->cku_err.re_status = RPC_CANTSEND;
869                 p->cku_err.re_errno = EIO;
870                 goto done;
871         }

873         /*
874          * Setup RDMA WRITE chunk list for nfs read operation
875          * other operations will have a NULL which will result
876          * as a NULL list in the XDR stream.
877          */
878         status = clnt_setup_wlist(conn, rdmahdr_o_xdrs, call_xdrp, &rndup);
879         if (status != CLNT_RDMA_SUCCESS) {
880                 rdma_buf_free(conn, &clmsg);
881                 p->cku_err.re_status = RPC_CANTSEND;
```



```
882                 p->cku_err.re_errno = EIO;
883                 goto done;
884         }

886         /*
887          * If NULL call and RPCSEC_GSS, provide a chunk such that
888          * large responses can flow back to the client.
889          * If RPCSEC_GSS with integrity or privacy is in use, get chunk.
890          */
891         if ((procnum == 0 && rpcsec_gss == TRUE) ||
892             (rpcsec_gss == TRUE && gss_i_or_p == TRUE))
893                 long_reply_len += 1024;

895         status = clnt_setup_long_reply(conn, &cl_long_reply, long_reply_len);

897         DTRACE_PROBE2(krpc__i__clntrdma__callit__longreply, int, status,
898             int, long_reply_len);

900         if (status != CLNT_RDMA_SUCCESS) {
901                 rdma_buf_free(conn, &clmsg);
902                 p->cku_err.re_status = RPC_CANTSEND;
903                 p->cku_err.re_errno = EIO;
904                 goto done;
905         }

907         /*
908          * XDR encode the RDMA_REPLY write chunk
909          */
910         seg_array_len = (cl_long_reply ? 1 : 0);
911         (void) xdr_encode_reply_wchunk(rdmahdr_o_xdrs, cl_long_reply,
912             seg_array_len);

914         /*
915          * Construct a clist in "sendlist" that represents what we
916          * will push over the wire.
917          *
918          * Start with the RDMA header and clist (if any)
919          */
920         clist_add(&cl_sendlist, 0, XDR_GETPOS(rdmahdr_o_xdrs), &clmsg.handle,
921             clmsg.addr, NULL, NULL);

923         /*
924          * Put the RPC call message in  sendlist if small RPC
925          */
926         if (op == RDMA_MSG) {
927                 clist_add(&cl_sendlist, 0, p->cku_outsz, &rpcmsg.handle,
928                     rpcmsg.addr, NULL, NULL);
929         } else {
930                 /* Long RPC already in chunk list */
931                 RCSTAT_INCR(rclongrpcs);
932         }

934         /*
935          * Set up a reply buffer ready for the reply
936          */
937         status = rdma_clnt_postrecv(conn, p->cku_xid);
938         if (status != RDMA_SUCCESS) {
939                 rdma_buf_free(conn, &clmsg);
940                 p->cku_err.re_status = RPC_CANTSEND;
941                 p->cku_err.re_errno = EIO;
942                 goto done;
943         }

945         /*
946          * sync the memory for dma
947          */
```

```
948         if (cl != NULL) {
949                 status = clist_syncmem(conn, cl, CLIST_REG_SOURCE);
950                 if (status != RDMA_SUCCESS) {
951                         (void) rdma_clnt_postrecv_remove(conn, p->cku_xid);
952                         rdma_buf_free(conn, &clmsg);
953                         p->cku_err.re_status = RPC_CANTSEND;
954                         p->cku_err.re_errno = EIO;
955                         goto done;
956                 }
957         }

959         /*
960          * Send the RDMA Header and RPC call message to the server
961          */
962         status = RDMA_SEND(conn, cl_sendlist, p->cku_xid);
963         if (status != RDMA_SUCCESS) {
964                 (void) rdma_clnt_postrecv_remove(conn, p->cku_xid);
965                 p->cku_err.re_status = RPC_CANTSEND;
966                 p->cku_err.re_errno = EIO;
967                 goto done;
968         }

970         /*
971          * RDMA plugin now owns the send msg buffers.
972          * Clear them out and don't free them.
973          */
974         clmsg.addr = NULL;
975         if (rpcmsg.type == SEND_BUFFER)
976                 rpcmsg.addr = NULL;

978         /*
979          * Recv rpc reply
980          */
981         status = RDMA_RECV(conn, &cl_recvlist, p->cku_xid);

983         /*
984          * Now check recv status
985          */
986         if (status != 0) {
987                 if (status == RDMA_INTR) {
988                         p->cku_err.re_status = RPC_INTR;
989                         p->cku_err.re_errno = EINTR;
990                         RCSTAT_INCR(rcintrs);
991                 } else if (status == RPC_TIMEDOUT) {
992                         p->cku_err.re_status = RPC_TIMEDOUT;
993                         p->cku_err.re_errno = ETIMEDOUT;
994                         RCSTAT_INCR(rctimeouts);
995                 } else {
996                         p->cku_err.re_status = RPC_CANTRECV;
997                         p->cku_err.re_errno = EIO;
998                 }
999                 goto done;
1000        }

1002        /*
1003         * Process the reply message.
1004         *
1005         * First the chunk list (if any)
1006         */
1007        rdmahdr_i_xdrs = &(p->cku_inxdr);
1008        xdrmem_create(rdmahdr_i_xdrs,
1009            (caddr_t)(uintptr_t)cl_recvlist->w.c_saddr3,
1010            cl_recvlist->c_len, XDR_DECODE);

1012        /*
1013         * Treat xid as opaque (xid is the first entity
```



```
1014         * in the rpc rdma message).
1015         * Skip xid and set the xdr position accordingly.
1016         */
1017        XDR_SETPOS(rdmahdr_i_xdrs, sizeof (uint32_t));
1018        (void) xdr_u_int(rdmahdr_i_xdrs, &vers);
1019        (void) xdr_u_int(rdmahdr_i_xdrs, &rdma_credit);
1020        (void) xdr_u_int(rdmahdr_i_xdrs, &op);
1021        (void) xdr_do_clist(rdmahdr_i_xdrs, &cl);

1023        clnt_update_credit(conn, rdma_credit);

1025        wlist_exists_reply = FALSE;
1026        if (! xdr_decode_wlist(rdmahdr_i_xdrs, &cl_rpcreply_wlist,
1027            &wlist_exists_reply)) {
1028                DTRACE_PROBE(krpc__e__clntrdma__callit__wlist_decode);
1029                p->cku_err.re_status = RPC_CANTDECODERES;
1030                p->cku_err.re_errno = EIO;
1031                goto done;
1032        }

1034        /*
1035         * The server shouldn't have sent a RDMA_SEND that
1036         * the client needs to RDMA_WRITE a reply back to
1037         * the server.  So silently ignoring what the
1038         * server returns in the rdma_reply section of the
1039         * header.
1040         */
1041        (void) xdr_decode_reply_wchunk(rdmahdr_i_xdrs, &cl_rdma_reply);
1042        off = xdr_getpos(rdmahdr_i_xdrs);

1044        clnt_decode_long_reply(conn, cl_long_reply,
1045            cl_rdma_reply, &replyxdr, &reply_xdrp,
1046            cl, cl_recvlist, op, off);

1048        if (reply_xdrp == NULL)
1049                goto done;

1051        if (wlist_exists_reply) {
1052                XDR_CONTROL(reply_xdrp, XDR_RDMA_SET_WLIST, cl_rpcreply_wlist);
1053        }

1055        reply_msg.rm_direction = REPLY;
1056        reply_msg.rm_reply.rp_stat = MSG_ACCEPTED;
1057        reply_msg.acpted_rply.ar_stat = SUCCESS;
1058        reply_msg.acpted_rply.ar_verf = _null_auth;

1060        /*
1061         *  xdr_results will be done in AUTH_UNWRAP.
1062         */
1063        reply_msg.acpted_rply.ar_results.where = NULL;
1064        reply_msg.acpted_rply.ar_results.proc = xdr_void;

1066        /*
1067         * Decode and validate the response.
1068         */
1069        if (xdr_replymsg(reply_xdrp, &reply_msg)) {
1070                enum clnt_stat re_status;

1072                _seterr_reply(&reply_msg, &(p->cku_err));

1074                re_status = p->cku_err.re_status;
1075                if (re_status == RPC_SUCCESS) {
1076                        /*
1077                         * Reply is good, check auth.
1078                         */
1079                        if (!AUTH_VALIDATE(h->cl_auth,
```

```
1080                            &reply_msg.acpted_rply.ar_verf)) {
1081                                p->cku_err.re_status = RPC_AUTHERROR;
1082                                p->cku_err.re_why = AUTH_INVALIDRESP;
1083                                RCSTAT_INCR(rcbadverfs);
1084                                DTRACE_PROBE(
1085                                    krpc__e__clntrdma__callit__authvalidate);
1086                        } else if (!AUTH_UNWRAP(h->cl_auth, reply_xdrp,
1087                            xdr_results, resultsp)) {
1088                                p->cku_err.re_status = RPC_CANTDECODERES;
1089                                p->cku_err.re_errno = EIO;
1090                                DTRACE_PROBE(
1091                                    krpc__e__clntrdma__callit__authunwrap);
1092                        }
1093                } else {
1094                        /* set errno in case we can't recover */
1095                        if (re_status != RPC_VERSMISMATCH &&
1096                            re_status != RPC_AUTHERROR &&
1097                            re_status != RPC_PROGVERSMISMATCH)
1098                                p->cku_err.re_errno = EIO;
1099
1100                        if (re_status == RPC_AUTHERROR) {
1101                                if ((refresh_attempt > 0) &&
1102                                    AUTH_REFRESH(h->cl_auth, &reply_msg,
1103                                    p->cku_cred)) {
1104                                        refresh_attempt--;
1105                                        try_call_again = 1;
1106                                        goto done;
1107                                }
1108
1109                                try_call_again = 0;
1110
1111                                /*
1112                                 * We have used the client handle to
1113                                 * do an AUTH_REFRESH and the RPC status may
1114                                 * be set to RPC_SUCCESS; Let's make sure to
1115                                 * set it to RPC_AUTHERROR.
1116                                 */
1117                                p->cku_err.re_status = RPC_AUTHERROR;
1118
1119                                /*
1120                                 * Map recoverable and unrecoverable
1121                                 * authentication errors to appropriate
1122                                 * errno
1123                                 */
1124                                switch (p->cku_err.re_why) {
1125                                case AUTH_BADCRED:
1126                                case AUTH_BADVERF:
1127                                case AUTH_INVALIDRESP:
1128                                case AUTH_TOOWEAK:
1129                                case AUTH_FAILED:
1130                                case RPCSEC_GSS_NOCRED:
1131                                case RPCSEC_GSS_FAILED:
1132                                        p->cku_err.re_errno = EACCES;
1133                                        break;
1134                                case AUTH_REJECTEDCRED:
1135                                case AUTH_REJECTEDVERF:
1136                                default:
1137                                        p->cku_err.re_errno = EIO;
1138                                        break;
1139                                }
1140                        }
1141                        DTRACE_PROBE1(krpc__e__clntrdma__callit__rpcfailed,
1142                            int, p->cku_err.re_why);
1143                }
1144        } else {
1145                p->cku_err.re_status = RPC_CANTDECODERES;
```



```
1146                p->cku_err.re_errno = EIO;
1147                DTRACE_PROBE(krpc__e__clntrdma__callit__replymsg);
1148        }

1150 done:
1151        clnt_return_credit(conn);

1153        if (cl_sendlist != NULL)
1154                clist_free(cl_sendlist);

1156        /*
1157         * If rpc reply is in a chunk, free it now.
1158         */
1159        if (cl_long_reply) {
1160                (void) clist_deregister(conn, cl_long_reply);
1161                rdma_buf_free(conn, &cl_long_reply->rb_longbuf);
1162                clist_free(cl_long_reply);
1163        }

1165        if (call_xdrp)
1166                XDR_DESTROY(call_xdrp);

1168        if (rndup.rb_private) {
1169                rdma_buf_free(conn, &rndup);
1170        }

1172        if (reply_xdrp) {
1173                (void) xdr_rpc_free_verifier(reply_xdrp, &reply_msg);
1174                XDR_DESTROY(reply_xdrp);
1175        }

1177        if (cl_rdma_reply) {
1178                clist_free(cl_rdma_reply);
1179        }

1181        if (cl_recvlist) {
1182                rdma_buf_t      recvmsg = {0};
1183                recvmsg.addr = (caddr_t)(uintptr_t)cl_recvlist->w.c_saddr3;
1184                recvmsg.type = RECV_BUFFER;
1185                RDMA_BUF_FREE(conn, &recvmsg);
1186                clist_free(cl_recvlist);
1187        }

1189        RDMA_REL_CONN(conn);

1191        if (try_call_again)
1192                goto call_again;

1194        if (p->cku_err.re_status != RPC_SUCCESS) {
1195                RCSTAT_INCR(rcbadcalls);
1196        }
1197        return (p->cku_err.re_status);
1198 }
```

unchanged_portion_omitted_

```
**********************************************************
  134428 Wed Aug 19 07:25:22 2015
new/usr/src/uts/common/rpc/rpcib.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

2172 /*
2173  * Wait for send completion notification. Only on receiving a
2174  * notification be it a successful or error completion, free the
2175  * send_wid.
2176  */
2177 static rdma_stat
2178 rib_sendwait(rib_qp_t *qp, struct send_wid *wd)
2179 {
2180         clock_t timout, cv_wait_ret;
2181         rdma_stat error = RDMA_SUCCESS;
2182         int     i;

2184         /*
2185          * Wait for send to complete
2186          */
2187         ASSERT(wd != NULL);
2188         mutex_enter(&wd->sendwait_lock);
2189         if (wd->status == (uint_t)SEND_WAIT) {
2190                 timout = drv_sectohz(SEND_WAIT_TIME) +
2190                 timout = drv_usectohz(SEND_WAIT_TIME * 1000000) +
2191                     ddi_get_lbolt();

2193                 if (qp->mode == RIB_SERVER) {
2194                         while ((cv_wait_ret = cv_timedwait(&wd->wait_cv,
2195                             &wd->sendwait_lock, timout)) > 0 &&
2196                             wd->status == (uint_t)SEND_WAIT)
2197                                 ;
2198                         switch (cv_wait_ret) {
2199                         case -1:        /* timeout */
2200                                 DTRACE_PROBE(rpcib__i__srvsendwait__timeout);

2202                                 wd->cv_sig = 0;         /* no signal needed */
2203                                 error = RDMA_TIMEDOUT;
2204                                 break;
2205                         default:        /* got send completion */
2206                                 break;
2207                         }
2208                 } else {
2209                         while ((cv_wait_ret = cv_timedwait_sig(&wd->wait_cv,
2210                             &wd->sendwait_lock, timout)) > 0 &&
2211                             wd->status == (uint_t)SEND_WAIT)
2212                                 ;
2213                         switch (cv_wait_ret) {
2214                         case -1:        /* timeout */
2215                                 DTRACE_PROBE(rpcib__i__clntsendwait__timeout);

2217                                 wd->cv_sig = 0;         /* no signal needed */
2218                                 error = RDMA_TIMEDOUT;
2219                                 break;
2220                         case 0:         /* interrupted */
2221                                 DTRACE_PROBE(rpcib__i__clntsendwait__intr);

2223                                 wd->cv_sig = 0;         /* no signal needed */
2224                                 error = RDMA_INTR;
2225                                 break;
2226                         default:        /* got send completion */
2227                                 break;
2228                         }
2229                 }
```



```
2230         }

2232         if (wd->status != (uint_t)SEND_WAIT) {
2233                 /* got send completion */
2234                 if (wd->status != RDMA_SUCCESS) {
2235                         switch (wd->status) {
2236                         case RDMA_CONNLOST:
2237                                 error = RDMA_CONNLOST;
2238                                 break;
2239                         default:
2240                                 error = RDMA_FAILED;
2241                                 break;
2242                         }
2243                 }
2244                 for (i = 0; i < wd->nsbufs; i++) {
2245                         rib_rbuf_free(qptoc(qp), SEND_BUFFER,
2246                             (void *)(uintptr_t)wd->sbufaddr[i]);
2247                 }

2249                 rib_send_rele(qp);

2251                 mutex_exit(&wd->sendwait_lock);
2252                 (void) rib_free_sendwait(wd);

2254         } else {
2255                 mutex_exit(&wd->sendwait_lock);
2256         }
2257         return (error);
2258 }
______unchanged_portion_omitted_

2414 /*
2415  * Deprecated/obsolete interface not used currently
2416  * but earlier used for READ-READ protocol.
2417  * Send RPC reply and wait for RDMA_DONE.
2418  */
2419 rdma_stat
2420 rib_send_resp(CONN *conn, struct clist *cl, uint32_t msgid)
2421 {
2422         rdma_stat ret = RDMA_SUCCESS;
2423         struct rdma_done_list *rd;
2424         clock_t cv_wait_ret;
2425         caddr_t *wid = NULL;
2426         rib_qp_t *qp = ctoqp(conn);

2428         mutex_enter(&qp->rdlist_lock);
2429         rd = rdma_done_add(qp, msgid);

2431         /* No cv_signal (whether send-wait or no-send-wait) */
2432         ret = rib_send_and_wait(conn, cl, msgid, 1, 0, wid);

2434         if (ret != RDMA_SUCCESS) {
2435                 rdma_done_rm(qp, rd);
2436         } else {
2437                 /*
2438                  * Wait for RDMA_DONE from remote end
2439                  */
2440                 cv_wait_ret = cv_reltimedwait(&rd->rdma_done_cv,
2441                     &qp->rdlist_lock, drv_sectohz(REPLY_WAIT_TIME),
2441                     &qp->rdlist_lock, drv_usectohz(REPLY_WAIT_TIME * 1000000),
2442                     TR_CLOCK_TICK);

2444                 rdma_done_rm(qp, rd);

2446                 if (cv_wait_ret < 0) {
2447                         ret = RDMA_TIMEDOUT;
```

```
2448                 }
2449         }

2451         mutex_exit(&qp->rdlist_lock);
2452         return (ret);
2453 }
_____unchanged_portion_omitted_

2661 /*
2662  * Client side only interface to "recv" the rpc reply buf
2663  * posted earlier by rib_post_resp(conn, cl, msgid).
2664  */
2665 rdma_stat
2666 rib_recv(CONN *conn, struct clist **clp, uint32_t msgid)
2667 {
2668         struct reply *rep = NULL;
2669         clock_t timout, cv_wait_ret;
2670         rdma_stat ret = RDMA_SUCCESS;
2671         rib_qp_t *qp = ctoqp(conn);

2673         /*
2674          * Find the reply structure for this msgid
2675          */
2676         mutex_enter(&qp->replylist_lock);

2678         for (rep = qp->replylist; rep != NULL; rep = rep->next) {
2679                 if (rep->xid == msgid)
2680                         break;
2681         }

2683         if (rep != NULL) {
2684                 /*
2685                  * If message not yet received, wait.
2686                  */
2687                 if (rep->status == (uint_t)REPLY_WAIT) {
2688                         timout = ddi_get_lbolt() +
2689                             drv_sectohz(REPLY_WAIT_TIME);
2689                             drv_usectohz(REPLY_WAIT_TIME * 1000000);

2691                         while ((cv_wait_ret = cv_timedwait_sig(&rep->wait_cv,
2692                             &qp->replylist_lock, timout)) > 0 &&
2693                             rep->status == (uint_t)REPLY_WAIT)
2694                                 ;

2696                         switch (cv_wait_ret) {
2697                         case -1:        /* timeout */
2698                                 ret = RDMA_TIMEDOUT;
2699                                 break;
2700                         case 0:
2701                                 ret = RDMA_INTR;
2702                                 break;
2703                         default:
2704                                 break;
2705                         }
2706                 }

2708                 if (rep->status == RDMA_SUCCESS) {
2709                         struct clist *cl = NULL;

2711                         /*
2712                          * Got message successfully
2713                          */
2714                         clist_add(&cl, 0, rep->bytes_xfer, NULL,
2715                             (caddr_t)(uintptr_t)rep->vaddr_cq, NULL, NULL);
2716                         *clp = cl;
2717                 } else {
```



```
2718                         if (rep->status != (uint_t)REPLY_WAIT) {
2719                                 /*
2720                                  * Got error in reply message. Free
2721                                  * recv buffer here.
2722                                  */
2723                                 ret = rep->status;
2724                                 rib_rbuf_free(conn, RECV_BUFFER,
2725                                     (caddr_t)(uintptr_t)rep->vaddr_cq);
2726                         }
2727                 }
2728                 (void) rib_remreply(qp, rep);
2729         } else {
2730                 /*
2731                  * No matching reply structure found for given msgid on the
2732                  * reply wait list.
2733                  */
2734                 ret = RDMA_INVAL;
2735                 DTRACE_PROBE(rpcib__i__nomatchxid2);
2736         }

2738         /*
2739          * Done.
2740          */
2741         mutex_exit(&qp->replylist_lock);
2742         return (ret);
2743 }
_____unchanged_portion_omitted_

4211 /*
4212  * rib_find_hca_connection
4213  *
4214  * if there is an existing connection to the specified address then
4215  * it will be returned in conn, otherwise conn will be set to NULL.
4216  * Also cleans up any connection that is in error state.
4217  */
4218 static int
4219 rib_find_hca_connection(rib_hca_t *hca, struct netbuf *s_svcaddr,
4220     struct netbuf *d_svcaddr, CONN **conn)
4221 {
4222         CONN *cn;
4223         clock_t cv_stat, timout;

4225         *conn = NULL;
4226 again:
4227         rw_enter(&hca->cl_conn_list.conn_lock, RW_READER);
4228         cn = hca->cl_conn_list.conn_hd;
4229         while (cn != NULL) {
4230                 /*
4231                  * First, clear up any connection in the ERROR state
4232                  */
4233                 mutex_enter(&cn->c_lock);
4234                 if (cn->c_state == C_ERROR_CONN) {
4235                         if (cn->c_ref == 0) {
4236                                 /*
4237                                  * Remove connection from list and destroy it.
4238                                  */
4239                                 cn->c_state = C_DISCONN_PEND;
4240                                 mutex_exit(&cn->c_lock);
4241                                 rw_exit(&hca->cl_conn_list.conn_lock);
4242                                 rib_conn_close((void *)cn);
4243                                 goto again;
4244                         }
4245                         mutex_exit(&cn->c_lock);
4246                         cn = cn->c_next;
4247                         continue;
4248                 }
```

```
4249                if (cn->c_state == C_DISCONN_PEND) {
4250                        mutex_exit(&cn->c_lock);
4251                        cn = cn->c_next;
4252                        continue;
4253                }

4255                /*
4256                 * source address is only checked for if there is one,
4257                 * this is the case for retries.
4258                 */
4259                if ((cn->c_raddr.len == d_svcaddr->len) &&
4260                    (bcmp(d_svcaddr->buf, cn->c_raddr.buf,
4261                    d_svcaddr->len) == 0) &&
4262                    ((s_svcaddr->len == 0) ||
4263                    ((cn->c_laddr.len == s_svcaddr->len) &&
4264                    (bcmp(s_svcaddr->buf, cn->c_laddr.buf,
4265                    s_svcaddr->len) == 0)))) {
4266                        /*
4267                         * Our connection. Give up conn list lock
4268                         * as we are done traversing the list.
4269                         */
4270                        rw_exit(&hca->cl_conn_list.conn_lock);
4271                        if (cn->c_state == C_CONNECTED) {
4272                                cn->c_ref++;    /* sharing a conn */
4273                                mutex_exit(&cn->c_lock);
4274                                *conn = cn;
4275                                return (RDMA_SUCCESS);
4276                        }
4277                        if (cn->c_state == C_CONN_PEND) {
4278                                /*
4279                                 * Hold a reference to this conn before
4280                                 * we give up the lock.
4281                                 */
4282                                cn->c_ref++;
4283                                timout =  ddi_get_lbolt() +
4284                                    drv_sectohz(CONN_WAIT_TIME);
4284                                    drv_usectohz(CONN_WAIT_TIME * 1000000);
4285                                while ((cv_stat = cv_timedwait_sig(&cn->c_cv,
4286                                    &cn->c_lock, timout)) > 0 &&
4287                                    cn->c_state == C_CONN_PEND)
4288                                        ;
4289                                if (cv_stat == 0) {
4290                                        (void) rib_conn_release_locked(cn);
4291                                        return (RDMA_INTR);
4292                                }
4293                                if (cv_stat < 0) {
4294                                        (void) rib_conn_release_locked(cn);
4295                                        return (RDMA_TIMEDOUT);
4296                                }
4297                                if (cn->c_state == C_CONNECTED) {
4298                                        *conn = cn;
4299                                        mutex_exit(&cn->c_lock);
4300                                        return (RDMA_SUCCESS);
4301                                } else {
4302                                        (void) rib_conn_release_locked(cn);
4303                                        return (RDMA_TIMEDOUT);
4304                                }
4305                        }
4306                }
4307                mutex_exit(&cn->c_lock);
4308                cn = cn->c_next;
4309        }
4310        rw_exit(&hca->cl_conn_list.conn_lock);
4311        *conn = NULL;
4312        return (RDMA_FAILED);
4313 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   39260 Wed Aug 19 07:25:22 2015
new/usr/src/uts/common/rpc/sec_gss/rpcsec_gss.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 538 /*
 539  * Private interface to create a context.  This is the interface
 540  * that's invoked when the context has to be refreshed.
 541  */
 542 static int
 543 rpc_gss_seccreate_pvt(gssstat, minor_stat, auth, ap, desired_mech_type,
 544                       actual_mech_type, ret_flags, time_rec, cr, isrefresh)
 545         OM_uint32               *gssstat;
 546         OM_uint32               *minor_stat;
 547         AUTH                    *auth;
 548         rpc_gss_data            *ap;
 549         gss_OID                 desired_mech_type;
 550         gss_OID                 *actual_mech_type;
 551         int                     *ret_flags;
 552         OM_uint32               *time_rec;
 553         cred_t                  *cr;
 554         int                     isrefresh;
 555 {
 556         CLIENT                  *clnt = ap->clnt;
 557         AUTH                    *save_auth;
 558         enum clnt_stat          callstat;
 559         rpc_gss_init_arg        call_arg;
 560         rpc_gss_init_res        call_res;
 561         gss_buffer_desc         *input_token_p, input_token, process_token;
 562         int                     free_results = 0;
 563         k_sigset_t              smask;
 564         int                     error = 0;

 566         /*
 567          * (re)initialize AUTH handle and private data.
 568          */
 569         bzero((char *)auth, sizeof (*auth));
 570         auth->ah_ops = &rpc_gss_ops;
 571         auth->ah_private = (caddr_t)ap;
 572         auth->ah_cred.oa_flavor = RPCSEC_GSS;

 574         ap->established = FALSE;
 575         ap->ctx_handle.length = 0;
 576         ap->ctx_handle.value = NULL;
 577         ap->context = NULL;
 578         ap->seq_num = 0;
 579         ap->gss_proc = RPCSEC_GSS_INIT;

 581         /*
 582          * should not change clnt->cl_auth at this time, so save
 583          * old handle
 584          */
 585         save_auth = clnt->cl_auth;
 586         clnt->cl_auth = auth;

 588         /*
 589          * set state for starting context setup
 590          */
 591         bzero((char *)&call_arg, sizeof (call_arg));
 592         input_token_p = GSS_C_NO_BUFFER;

 594 next_token:
 595         *gssstat = kgss_init_sec_context(minor_stat,
 596                                          ap->my_cred,
```



```
 597                                          &ap->context,
 598                                          ap->target_name,
 599                                          desired_mech_type,
 600                                          ap->req_flags,
 601                                          ap->time_req,
 602                                          NULL,
 603                                          input_token_p,
 604                                          actual_mech_type,
 605                                          &call_arg,
 606                                          ret_flags,
 607                                          time_rec,
 608                                          crgetuid(cr));

 610         if (input_token_p != GSS_C_NO_BUFFER) {
 611                 OM_uint32 minor_stat2;

 613                 (void) gss_release_buffer(&minor_stat2, input_token_p);
 614                 input_token_p = GSS_C_NO_BUFFER;
 615         }

 617         if (*gssstat != GSS_S_COMPLETE && *gssstat != GSS_S_CONTINUE_NEEDED) {
 618                 rpc_gss_display_status(*gssstat, *minor_stat,
 619                         desired_mech_type, crgetuid(cr),
 620                         "rpcsec_gss_secreate_pvt:gss_init_sec_context");
 621                 error = EACCES;
 622                 goto cleanup;
 623         }

 625         /*
 626          * if we got a token, pass it on
 627          */
 628         if (call_arg.length != 0) {
 629                 struct timeval timeout = {30, 0};
 630                 int      rpcsec_retry = isrefresh ?
 631                         RPCSEC_GSS_REFRESH_ATTEMPTS : 1;
 632                 uint32_t oldxid;
 633                 uint32_t zeroxid = 0;

 635                 bzero((char *)&call_res, sizeof (call_res));

 637                 (void) CLNT_CONTROL(clnt, CLGET_XID, (char *)&oldxid);
 638                 (void) CLNT_CONTROL(clnt, CLSET_XID, (char *)&zeroxid);


 641                 while (rpcsec_retry > 0) {
 642                         struct rpc_err rpcerr;

 644                         sigintr(&smask, INTERRUPT_OK);

 646                         callstat = clnt_call(clnt, NULLPROC,
 647                                 __xdr_rpc_gss_init_arg, (caddr_t)&call_arg,
 648                                 __xdr_rpc_gss_init_res, (caddr_t)&call_res,
 649                                 timeout);

 651                         sigunintr(&smask);

 653                         if (callstat == RPC_SUCCESS) {
 654                                 error = 0;
 655                                 if (isrefresh &&
 656                                     call_res.gss_major == GSS_S_FAILURE) {

 658                                         clock_t one_sec = drv_sectohz(1);
 658                                         clock_t one_sec = drv_usectohz(1000000);

 660                                         rpcsec_retry--;
```

```
662                                     /*
663                                      * Pause a little and try again.
664                                      */
666                                     if (clnt->cl_nosignal == TRUE) {
667                                             delay(one_sec);
668                                     } else {
669                                             if (delay_sig(one_sec)) {
670                                                     error = EINTR;
671                                                     break;
672                                             }
673                                     }
674                                     continue;
675                             }
676                             break;
677                     }
679                     if (callstat == RPC_TIMEDOUT) {
680                             error = ETIMEDOUT;
681                             break;
682                     }
684                     if (callstat == RPC_XPRTFAILED) {
685                             error = ECONNRESET;
686                             break;
687                     }
689                     if (callstat == RPC_INTR) {
690                             error = EINTR;
691                             break;
692                     }
694                     if (callstat == RPC_INPROGRESS) {
695                             continue;
696                     }
698                     clnt_geterr(clnt, &rpcerr);
699                     error = rpcerr.re_errno;
700                     break;
701             }
703             (void) CLNT_CONTROL(clnt, CLSET_XID, (char *)&oldxid);
705             (void) gss_release_buffer(minor_stat, &call_arg);
707             if (callstat != RPC_SUCCESS) {
708                     RPCGSS_LOG(1,
709                         "rpc_gss_seccreate_pvt: clnt_call failed %d\n",
710                         callstat);
711                     goto cleanup;
712             }
714             /*
715              * we have results - note that these need to be freed
716              */
717             free_results = 1;
719             if ((call_res.gss_major != GSS_S_COMPLETE) &&
720                 (call_res.gss_major != GSS_S_CONTINUE_NEEDED)) {
721                     RPCGSS_LOG1(1, "rpc_gss_seccreate_pvt: "
722                             "call_res gss_major %x, gss_minor %x\n",
723                             call_res.gss_major, call_res.gss_minor);
724                     error = EACCES;
725                     goto cleanup;
726             }
```



```
728             ap->gss_proc = RPCSEC_GSS_CONTINUE_INIT;
730             /*
731              * check for ctx_handle
732              */
733             if (ap->ctx_handle.length == 0) {
734                     if (call_res.ctx_handle.length == 0) {
735                             RPCGSS_LOG0(1, "rpc_gss_seccreate_pvt: zero "
736                                     "length handle in response\n");
737                             error = EACCES;
738                             goto cleanup;
739                     }
740                     GSS_DUP_BUFFER(ap->ctx_handle,
741                                     call_res.ctx_handle);
742             } else if (!GSS_BUFFERS_EQUAL(ap->ctx_handle,
743                                     call_res.ctx_handle)) {
744                     RPCGSS_LOG0(1,
745                     "rpc_gss_seccreate_pvt: ctx_handle not the same\n");
746                     error = EACCES;
747                     goto cleanup;
748             }
750             /*
751              * check for token
752              */
753             if (call_res.token.length != 0) {
754                     if (*gssstat == GSS_S_COMPLETE) {
755                             RPCGSS_LOG0(1, "rpc_gss_seccreate_pvt: non "
756                                     "zero length token in response, but "
757                                     "gsstat == GSS_S_COMPLETE\n");
758                             error = EACCES;
759                             goto cleanup;
760                     }
761                     GSS_DUP_BUFFER(input_token, call_res.token);
762                     input_token_p = &input_token;
764             } else if (*gssstat != GSS_S_COMPLETE) {
765                     RPCGSS_LOG0(1, "rpc_gss_seccreate_pvt:zero length "
766                             "token in response, but "
767                             "gsstat != GSS_S_COMPLETE\n");
768                     error = EACCES;
769                     goto cleanup;
770             }
772             /* save the sequence window value; validate later */
773             ap->seq_window = call_res.seq_window;
774             xdr_free(__xdr_rpc_gss_init_res, (caddr_t)&call_res);
775             free_results = 0;
776     }
778     /*
779      * results were okay.. continue if necessary
780      */
781     if (*gssstat == GSS_S_CONTINUE_NEEDED) {
782             goto next_token;
783     }
785     /*
786      * Context is established. Now use kgss_export_sec_context and
787      * kgss_import_sec_context to transfer the context from the user
788      * land to kernel if the mechanism specific kernel module is
789      * available.
790      */
791     *gssstat  = kgss_export_sec_context(minor_stat, ap->context,
792                                         &process_token);
793     if (*gssstat == GSS_S_NAME_NOT_MN) {
```

```
794                 RPCGSS_LOG(2, "rpc_gss_seccreate_pvt: export_sec_context "
795                         "Kernel Module unavailable  gssstat = 0x%x\n",
796                         *gssstat);
797                 goto done;
798         } else if (*gssstat != GSS_S_COMPLETE) {
799                 (void) rpc_gss_display_status(*gssstat, *minor_stat,
800                         isrefresh ? GSS_C_NULL_OID : *actual_mech_type,
801                                 crgetuid(cr),
802                         "rpcsec_gss_secreate_pvt:gss_export_sec_context");
803                 (void) kgss_delete_sec_context(minor_stat,
804                                 &ap->context, NULL);
805                 error = EACCES;
806                 goto cleanup;
807         } else if (process_token.length == 0) {
808                 RPCGSS_LOG0(1, "rpc_gss_seccreate_pvt:zero length "
809                                 "token in response for export_sec_context, but "
810                                 "gsstat == GSS_S_COMPLETE\n");
811                 (void) kgss_delete_sec_context(minor_stat,
812                                 &ap->context, NULL);
813                 error = EACCES;
814                 goto cleanup;
815         } else
816                 *gssstat = kgss_import_sec_context(minor_stat, &process_token,
817                                                    ap->context);

819         if (*gssstat == GSS_S_COMPLETE) {
820                 (void) gss_release_buffer(minor_stat, &process_token);
821         } else {
822                 rpc_gss_display_status(*gssstat, *minor_stat,
823                         desired_mech_type, crgetuid(cr),
824                         "rpcsec_gss_secreate_pvt:gss_import_sec_context");
825                 (void) kgss_delete_sec_context(minor_stat,
826                                 &ap->context, NULL);
827                 (void) gss_release_buffer(minor_stat, &process_token);
828                 error = EACCES;
829                 goto cleanup;
830         }

832 done:
833         /*
834          * Validate the sequence window - RFC 2203 section 5.2.3.1
835          */
836         if (!validate_seqwin(ap)) {
837                 error = EACCES;
838                 goto cleanup;
839         }

841         /*
842          * Done!  Security context creation is successful.
843          * Ready for exchanging data.
844          */
845         ap->established = TRUE;
846         ap->seq_num = 1;
847         ap->gss_proc = RPCSEC_GSS_DATA;
848         ap->invalid = FALSE;

850         clnt->cl_auth = save_auth;      /* restore cl_auth */

852         return (0);

854 cleanup:
855         if (free_results)
856                 xdr_free(__xdr_rpc_gss_init_res, (caddr_t)&call_res);
857         clnt->cl_auth = save_auth;      /* restore cl_auth */

859         /*
```



```
860          * If need to retry for AUTH_REFRESH, do not cleanup the
861          * auth private data.
862          */
863         if (isrefresh && (error == ETIMEDOUT || error == ECONNRESET)) {
864                 return (error);
865         }

867         if (ap->context != NULL) {
868                 rpc_gss_free_pvt(auth);
869         }

871         return (error? error : EACCES);
872 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10400 Wed Aug 19 07:25:22 2015
new/usr/src/uts/common/sys/bscv_impl.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 334 #define BSC_IMAGE_MAX_SIZE (0x20000 + sizeof (lom_prog_data_t))

 336 #define BSC_PROBE_FAULT_LIMIT   8       /* Tries before declaring lom dead */
 337 #define BSC_EVENT_POLL_NORMAL   drv_sectohz(1)
 338 #define BSC_EVENT_POLL_FAULTY   drv_sectohz(10)
 337 #define BSC_EVENT_POLL_NORMAL   (drv_usectohz(1000000))         /* 1 second */
 338 #define BSC_EVENT_POLL_FAULTY   (drv_usectohz(10000000))        /* 10 second */

 340 #define BSC_FAILURE_RETRY_LIMIT 5       /* Access retries before giving up */
 341 #define BSC_ERASE_RETRY_LIMIT   5       /* Erase retries */
 342 #define BSC_PAGE_RETRY_LIMIT    5       /* Page write retries */

 344 #define BSC_ADDR_CACHE_LIMIT    \
 345                 (sizeof (((bscv_soft_state_t *)NULL)->lom_regs))
 346 #define BSC_INFORM_ONLINE       0x4f530100
 347 #define BSC_INFORM_OFFLINE      0x4f530201
 348 #define BSC_INFORM_PANIC        0x4f530204

 350 #include <sys/lom_ebuscodes.h>

 352 typedef uint32_t bscv_addr_t;

 354 #define BSC_NEXUS_ADDR(ssp, chan, as, index) \
 355         (&((ssp)->channel[chan].regs[((as) * 256) + (index)]))

 357 #define BSC_NEXUS_OFFSET(as, index) (((as) * 256) + (index))

 359 #define BSCVA(as, index) (((as) * 256) + (index))

 361 #define PSR_SUCCESS(status)     (((status) & EBUS_PROGRAM_PSR_STATUS_MASK) == \
 362     EBUS_PROGRAM_PSR_SUCCESS)

 364 #define PSR_PROG(status)        (((status) & EBUS_PROGRAM_PSR_PROG_MODE) != 0)
 365 #ifdef  __cplusplus
 366 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4943 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/ddi.h
XXXX introduce drv_sectohz
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2014 Garrett D'Amore <garrett@damore.org>
 * Copyright 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 *
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


#ifndef _SYS_DDI_H
#define _SYS_DDI_H

#include <sys/types.h>
#include <sys/map.h>
#include <sys/buf.h>
#include <sys/uio.h>
#include <sys/stream.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * ddi.h -- the flag and function definitions needed by DDI-conforming
 * drivers.  This header file contains #undefs to undefine macros that
 * drivers would otherwise pick up in order that function definitions
 * may be used. Programmers should place the include of "sys/ddi.h"
 * after any header files that define the macros #undef'ed or the code
 * may compile incorrectly.
 */

/*
 * define min() and max() as macros so that drivers will not pick up the
 * min() and max() kernel functions since they do signed comparison only.
 */
#ifdef  min
```



```
#undef  min
#endif  /* min */
#define min(a, b)       ((a) < (b) ? (a) : (b))

#ifdef  max
#undef  max
#endif  /* max */
#define max(a, b)       ((a) < (b) ? (b) : (a))

#define TIME    1
#define UPROCP  2
#define PPGRP   3
#define LBOLT   4
#define SYSRINT 5
#define SYSXINT 6
#define SYSMINT 7
#define SYSRAWC 8
#define SYSCANC 9
#define SYSOUTC 10
#define PPID    11
#define PSID    12
#define UCRED   13

extern int drv_getparm(uint_t, void *);
extern int drv_setparm(uint_t, ulong_t);
extern void drv_usecwait(clock_t);
extern clock_t drv_hztousec(clock_t);
extern clock_t drv_usectohz(clock_t);
extern clock_t drv_sectohz(clock_t);
#endif /* ! codereview */
extern void delay(clock_t);
extern void time_to_wait(clock_t *, clock_t);

/* XXX -- should be changed to major_t */
/* convert external to internal major number */

extern int etoimajor(major_t);
/* convert internal to extern major number */
extern int itoemajor(major_t, int);
extern int drv_priv(struct cred *);

/*
 * The following declarations take the place of macros in
 * sysmacros.h The undefs are for any case where a driver includes
 * sysmacros.h, even though DDI conforming drivers must not.
 */
#undef getemajor
#undef geteminor
#undef getmajor
#undef getminor
#undef makedevice
#undef cmpdev
#undef expdev


extern major_t getemajor(dev_t);
extern minor_t geteminor(dev_t);
extern major_t getmajor(dev_t);
extern minor_t getminor(dev_t);
extern dev_t makedevice(major_t, minor_t);
extern o_dev_t cmpdev(dev_t);
extern dev_t expdev(dev_t);

/*
 * The following macros from param.h are also being converted to
 * functions and #undefs must be done here as well since param.h
```

```
 * will be included by most if not every driver
 */

#undef btop
#undef btopr
#undef ptob

extern unsigned long btop(unsigned long);
extern unsigned long btopr(unsigned long);
extern unsigned long ptob(unsigned long);


/* STREAMS drivers and modules must include stream.h to pick up the */
/* needed structure and flag definitions. As was the case with map.h, */
/* macros used by both the kernel and drivers in times past now have */
/* a macro definition for the kernel and a function definition for */
/* drivers. The following #undefs allow drivers to include stream.h */
/* but call the functions rather than macros. */

#undef OTHERQ
#undef RD
#undef WR
#undef SAMESTR
#undef datamsg

extern struct queue *OTHERQ(queue_t *); /* stream.h */
extern struct queue *RD(queue_t *);
extern struct queue *WR(queue_t *);
extern int SAMESTR(queue_t *);
extern int datamsg(unsigned char);

/* declarations of functions for allocating and deallocating the space */
/* for a buffer header (just a header, not the associated buffer) */

extern struct buf *getrbuf(int);
extern void freerbuf(struct buf *);

#ifdef  _KERNEL
/*
 * SVR4MP replacement for hat_getkpfnum()
 */
#define NOPAGE  (-1)    /* value returned for invalid addresses */

typedef pfn_t   ppid_t; /* a 'physical page identifier' - no math allowed! */

extern ppid_t kvtoppid(caddr_t);

extern int qassociate(queue_t *, int);

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_DDI_H */
```

```
**********************************************************
    4878 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/dktp/dadk.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

  70 #define DAD_SECSIZ      dad_phyg.g_secsiz

  72 /*
  73  * Local definitions, for clarity of code
  74  */

  76 /*
  77  * Parameters
  78  */
  79 #define DADK_BSY_TIMEOUT        drv_sectohz(5)
  79 #define DADK_BSY_TIMEOUT        (drv_usectohz(5 * 1000000))
  80 #define DADK_IO_TIME            35
  81 #define DADK_FLUSH_CACHE_TIME   60
  82 #define DADK_RETRY_COUNT        5
  83 #define DADK_SILENT             1

  85 #define PKT2DADK(pktp)  ((struct dadk *)(pktp)->cp_dev_private)

  87 /*
  88  * packet action codes
  89  */
  90 #define COMMAND_DONE            0
  91 #define COMMAND_DONE_ERROR      1
  92 #define QUE_COMMAND             2
  93 #define QUE_SENSE               3
  94 #define JUST_RETURN             4

  96 typedef struct  dadk_errstats {
  97         kstat_named_t dadk_softerrs;            /* Collecting Softerrs */
  98         kstat_named_t dadk_harderrs;            /* Collecting harderrs */
  99         kstat_named_t dadk_transerrs;           /* Collecting Transfer errs */
 100         kstat_named_t dadk_model;               /* model # of the disk */
 101         kstat_named_t dadk_revision;            /* The disk revision */
 102         kstat_named_t dadk_serial;              /* The disk serial number */
 103         kstat_named_t dadk_capacity;            /* Capacity of the disk */
 104         kstat_named_t dadk_rq_media_err;        /* Any media err seen */
 105         kstat_named_t dadk_rq_ntrdy_err;        /* Not ready errs */
 106         kstat_named_t dadk_rq_nodev_err;        /* No device errs */
 107         kstat_named_t dadk_rq_recov_err;        /* Recovered errs */
 108         kstat_named_t dadk_rq_illrq_err;        /* Illegal requests */
 109 } dadk_errstats_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   10853 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/fcoe/fcoe_common.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 362 /*
 363  * FCOE project global functions
 364  */
 365 #if !defined(__FUNCTION__)
 366 #define __FUNCTION__ ((caddr_t)__func__)
 367 #endif

 369 #define FCOE_STR_LEN 32

 371 /*
 372  * timestamp (golbal variable in sys/systm.h)
 373  */
 374 #define CURRENT_CLOCK           (ddi_get_lbolt())
 375 #define FCOE_SEC2TICK(x_sec)    drv_sectohz(x_sec)
 375 #define FCOE_SEC2TICK(x_sec)    (drv_usectohz((x_sec) * 1000000))

 377 /*
 378  * Form/convert mod_hash_key from/to xch ID
 379  */
 380 #define FMHK(x_xid)             (mod_hash_key_t)(uintptr_t)(x_xid)
 381 #define CMHK(x_key)             (uint16_t)(uintptr_t)(x_key)

 383 typedef void (*TQ_FUNC_P)(void *);
 384 extern void fcoe_trace(caddr_t ident, const char *fmt, ...);

 386 #endif

 388 #ifdef  __cplusplus
 389 }
______unchanged_portion_omitted_
```

```
**********************************************************
   26400 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/fibre-channel/fca/qlge/qlge.h
XXXX introduce drv_sectohz
**********************************************************
```

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

/*
 * Copyright 2010 QLogic Corporation. All rights reserved.
 */

#ifndef _QLGE_H
#define _QLGE_H

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunmdi.h>
#include <sys/modctl.h>
#include <sys/pci.h>
#include <sys/dlpi.h>
#include <sys/sdt.h>
#include <sys/mac_provider.h>
#include <sys/mac.h>
#include <sys/mac_flow.h>
#include <sys/mac_ether.h>
#include <sys/vlan.h>
#include <sys/netlb.h>
#include <sys/kmem.h>
#include <sys/file.h>
#include <sys/proc.h>
#include <sys/callb.h>
#include <sys/disp.h>
#include <sys/strsun.h>
#include <sys/ethernet.h>
#include <sys/miiregs.h>
#include <sys/kstat.h>
#include <sys/byteorder.h>
#include <sys/ddifm.h>
#include <sys/fm/protocol.h>
#include <sys/fm/util.h>
#include <sys/fm/io/ddi.h>

#include <qlge_hw.h>
```



```
#include <qlge_dbg.h>
#include <qlge_open.h>

#define ADAPTER_NAME            "qlge"

/*
 * Local Macro Definitions.
 */
#ifdef  TRUE
#undef  TRUE
#endif
#define TRUE    1

#ifdef  FALSE
#undef  FALSE
#endif
#define FALSE   0

/* #define QLGE_TRACK_BUFFER_USAGE */
/*
 * byte order, sparc is big endian, x86 is little endian,
 * but PCI is little endian only
 */
#ifdef sparc
#define cpu_to_le64(x)  BSWAP_64(x)
#define cpu_to_le32(x)  BSWAP_32(x)
#define cpu_to_le16(x)  BSWAP_16(x)
#define le64_to_cpu(x)  cpu_to_le64(x)
#define le32_to_cpu(x)  cpu_to_le32(x)
#define le16_to_cpu(x)  cpu_to_le16(x)
#else
#define cpu_to_le64(x)  (x)
#define cpu_to_le32(x)  (x)
#define cpu_to_le16(x)  (x)
#define le64_to_cpu(x)  (x)
#define le32_to_cpu(x)  (x)
#define le16_to_cpu(x)  (x)
#endif

/*
 * Macros to help code, maintain, etc.
 */

#define LSB(x)                  (uint8_t)(x)
#define MSB(x)                  (uint8_t)((uint16_t)(x) >> 8)

#define MSW(x)                  (uint16_t)((uint32_t)(x) >> 16)
#define LSW(x)                  (uint16_t)(x)

#define MS32(x)                 (uint32_t)((uint32_t)(x) >> 32)
#define LS32(x)                 (uint32_t)(x)

#define MSW_LSB(x)              (uint8_t)(LSB(MSW(x)))
#define MSW_MSB(x)              (uint8_t)(MSB(MSW(x)))

#define LSD(x)                  (uint32_t)(x)
#define MSD(x)                  (uint32_t)((uint64_t)(x) >> 32)

#define SHORT_TO_LONG(a, b)     (uint32_t)((uint16_t)b << 16 | (uint16_t)a)
#define CHAR_TO_SHORT(a, b)     (uint16_t)((uint8_t)b << 8 | (uint8_t)a)

#define SWAP_ENDIAN_16(x)       ((LSB(x) << 8) | MSB(x))

#define SWAP_ENDIAN_32(x)       ((SWAP_ENDIAN_16(LSW(x)) << 16) | \
                                    SWAP_ENDIAN_16(MSW(x)))
```

```
128 #define SWAP_ENDIAN_64(x)       ((SWAP_ENDIAN_32(LS32(x)) << 32) | \
129                                     SWAP_ENDIAN_32(MS32(x)))

131 #define QL_MIN(x, y)            ((x < y) ? x : y)

133 #define CARRIER_ON(qlge)        mac_link_update((qlge)->mh, LINK_STATE_UP)
134 #define CARRIER_OFF(qlge)       mac_link_update((qlge)->mh, LINK_STATE_DOWN)

136 /*
137  * qlge local function return status codes
138  */
139 #define QL_ERROR                1
140 #define QL_SUCCESS              0
141 /*
142  * Solaris version compatibility definitions.
143  */
144 #define QL_GET_LBOLT(timer)     timer = ddi_get_lbolt()
145 #define QL_DMA_XFER_COUNTER     (uint64_t)0xffffffff
146 #define QL_DRIVER_NAME(dip)     ddi_driver_name(ddi_get_parent(dip))

148 #define MINOR_NODE_FLAG         8

150 /*
151  * Host adapter default definitions.
152  */

154 /* Timeout timer counts in seconds (must greater than 1 second). */
155 #define USEC_PER_TICK           drv_hztousec(1)
156 #define TICKS_PER_SEC           drv_sectohz(1)
156 #define TICKS_PER_SEC           drv_usectohz(1000000)
157 #define QL_ONE_SEC_DELAY        1000000
158 #define QL_ONE_MSEC_DELAY       1000
159 #define TX_TIMEOUT              3*TICKS_PER_SEC
160 /*
161  * DMA attributes definitions.
162  */
163 #define QL_DMA_LOW_ADDRESS              (uint64_t)0
164 #define QL_DMA_HIGH_64BIT_ADDRESS       (uint64_t)0xffffffffffffffffull
165 #define QL_DMA_HIGH_32BIT_ADDRESS       (uint64_t)0xffffffff
166 #define QL_DMA_ADDRESS_ALIGNMENT        (uint64_t)8
167 #define QL_DMA_ALIGN_8_BYTE_BOUNDARY    (uint64_t)BIT_3
168 #define QL_DMA_RING_ADDRESS_ALIGNMENT   (uint64_t)64
169 #define QL_DMA_ALIGN_64_BYTE_BOUNDARY   (uint64_t)BIT_6
170 #define QL_DMA_BURSTSIZES               0xfff
171 #define QL_DMA_MIN_XFER_SIZE            1
172 #define QL_DMA_MAX_XFER_SIZE            (uint64_t)0xffffffff
173 #define QL_DMA_SEGMENT_BOUNDARY         (uint64_t)0xffffffff
174 #define QL_DMA_GRANULARITY              1
175 #define QL_DMA_XFER_FLAGS               0
176 #define QL_MAX_COOKIES                  16

178 /*
179  * ISP PCI Configuration.
180  */
181 #define QL_INTR_INTERVAL        128     /* default interrupt interval 128us */
182 #define QL_INTR_PKTS            8       /* default packet count threshold 8us */

184 /* GLD */
185 #define QL_STREAM_OPS(dev_ops, attach, detach)  \
186         DDI_DEFINE_STREAM_OPS(dev_ops, nulldev, nulldev, attach, detach, \
187             nodev, NULL, D_MP, NULL, ql_quiesce)

189 #define QL_GET_DEV(dip)         ((qlge_t *)(ddi_get_driver_private(dip)))
190 #define RESUME_TX(tx_ring)              mac_tx_update(tx_ring->qlge->mh);
191 #define RX_UPSTREAM(rx_ring, mp)        mac_rx(rx_ring->qlge->mh, \
192                                             rx_ring->qlge->handle, mp);
```



```
194 /* GLD DMA */
195 extern ddi_device_acc_attr_t ql_dev_acc_attr;
196 extern ddi_device_acc_attr_t ql_desc_acc_attr;
197 extern ddi_device_acc_attr_t ql_buf_acc_attr;

199 struct dma_info {
200         void            *vaddr;
201         ddi_dma_handle_t dma_handle;
202         ddi_acc_handle_t acc_handle;
203         uint64_t         dma_addr;
204         size_t           mem_len; /* allocated size */
205         offset_t         offset;  /* relative to handle */
206 };
```

*unchanged_portion_omitted_*

```
**********************************************************
   14509 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/ib/mgt/ibdm/ibdm_impl.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

338 #define IBDM_TIMEOUT_VALUE(t)   drv_sectohz(t)
338 #define IBDM_TIMEOUT_VALUE(t)   (drv_usectohz(t * 1000000))

340 #define IBDM_OUT_IBMFMSG_MADHDR(msg)\
341                 (msg->im_msgbufs_send.im_bufs_mad_hdr)

343 #define IBDM_IN_IBMFMSG_MADHDR(msg)\
344                 (msg->im_msgbufs_recv.im_bufs_mad_hdr)

346 #define IBDM_IN_IBMFMSG_STATUS(msg)\
347                 b2h16(msg->im_msgbufs_recv.im_bufs_mad_hdr->Status)

349 #define IBDM_IN_IBMFMSG_ATTR(msg)\
350                 b2h16(msg->im_msgbufs_recv.im_bufs_mad_hdr->AttributeID)

352 #define IBDM_IN_IBMFMSG_ATTRMOD(msg)\
353                 b2h32(msg->im_msgbufs_recv.im_bufs_mad_hdr->AttributeModifier)

355 #define IBDM_IN_IBMFMSG2IOU(msg)        (ib_dm_io_unitinfo_t *)\
356                 (msg->im_msgbufs_recv.im_bufs_cl_data)

358 #define IBDM_IN_IBMFMSG2IOC(msg)        (ib_dm_ioc_ctrl_profile_t *)\
359                 (msg->im_msgbufs_recv.im_bufs_cl_data)

361 #define IBDM_IN_IBMFMSG2SRVENT(msg)     (ib_dm_srv_t *)\
362                 (msg->im_msgbufs_recv.im_bufs_cl_data)

364 #define IBDM_IN_IBMFMSG2DIAGCODE(msg)   (uint32_t *)\
365                 (msg->im_msgbufs_recv.im_bufs_cl_data)

367 #define IBDM_GIDINFO2IOCINFO(gid_info, idx) \
368                 (ibdm_ioc_info_t *)&gid_info->gl_iou->iou_ioc_info[idx];

370 #define IBDM_IS_IOC_NUM_INVALID(ioc_no, gid_info)\
371                 ((ioc_no < 1) || (ioc_no >\
372                         gid_info->gl_iou->iou_info.iou_num_ctrl_slots))

374 #define IBDM_INVALID_PKEY(pkey) \
375                 (((pkey) == IB_PKEY_INVALID_FULL) || \
376                 ((pkey) == IB_PKEY_INVALID_LIMITED))

378 #ifdef DEBUG

380 void    ibdm_dump_mad_hdr(ib_mad_hdr_t *);
381 void    ibdm_dump_ibmf_msg(ibmf_msg_t *, int);
382 void    ibdm_dump_path_info(sa_path_record_t *);
383 void    ibdm_dump_classportinfo(ib_mad_classportinfo_t *);
384 void    ibdm_dump_iounitinfo(ib_dm_io_unitinfo_t *);
385 void    ibdm_dump_ioc_profile(ib_dm_ioc_ctrl_profile_t *);
386 void    ibdm_dump_service_entries(ib_dm_srv_t *);
387 void    ibdm_dump_sweep_fabric_timestamp(int);

389 #define ibdm_dump_mad_hdr(a)            ibdm_dump_mad_hdr(a)
390 #define ibdm_dump_ibmf_msg(a, b)        ibdm_dump_ibmf_msg(a, b)
391 #define ibdm_dump_path_info(a)          ibdm_dump_path_info(a)
392 #define ibdm_dump_classportinfo(a)      ibdm_dump_classportinfo(a)
393 #define ibdm_dump_iounitinfo(a)         ibdm_dump_iounitinfo(a)
394 #define ibdm_dump_ioc_profile(a)        ibdm_dump_ioc_profile(a)
395 #define ibdm_dump_service_entries(a)    ibdm_dump_service_entries(a)
```



```
397 #else

399 #define ibdm_dump_mad_hdr(a)
400 #define ibdm_dump_ibmf_msg(a, b)
401 #define ibdm_dump_path_info(a)
402 #define ibdm_dump_classportinfo(a)
403 #define ibdm_dump_iounitinfo(a)
404 #define ibdm_dump_ioc_profile(a)
405 #define ibdm_dump_service_entries(a)
406 #define ibdm_dump_sweep_fabric_timestamp(a)

408 #endif

410 #ifdef __cplusplus
411 }
______unchanged_portion_omitted_
```

```
**********************************************************
   75132 Wed Aug 19 07:25:23 2015
new/usr/src/uts/common/sys/scsi/targets/sddef.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

1100 /*
1101  * Conditional set def
1102  */
1103 #define SD_CONDSET(a, b, c, d)                  \
1104         { \
1105         a->c = ((fi_ ## b)->c);                 \
1106         SD_INFO(SD_LOG_IOERR, un,               \
1107                         "sd_fault_injection:"   \
1108                         "setting %s to %d\n",   \
1109                         d, ((fi_ ## b)->c));    \
1110         }

1112 /* SD FaultInjection ioctls */
1113 #define SDIOC                   ('T'<<8)
1114 #define SDIOCSTART              (SDIOC|1)
1115 #define SDIOCSTOP               (SDIOC|2)
1116 #define SDIOCINSERTPKT  (SDIOC|3)
1117 #define SDIOCINSERTXB   (SDIOC|4)
1118 #define SDIOCINSERTUN   (SDIOC|5)
1119 #define SDIOCINSERTARQ  (SDIOC|6)
1120 #define SDIOCPUSH               (SDIOC|7)
1121 #define SDIOCRETRIEVE   (SDIOC|8)
1122 #define SDIOCRUN                (SDIOC|9)
1123 #endif

1125 #else

1127 #undef  SDDEBUG
1128 #define SD_ERROR        { if (0) sd_log_err; }
1129 #define SD_TRACE        { if (0) sd_log_trace; }
1130 #define SD_INFO         { if (0) sd_log_info; }
1131 #define SD_DUMP_MEMORY  { if (0) sd_dump_memory; }
1132 #endif


1135 /*
1136  * Miscellaneous macros
1137  */

1139 #define SD_USECTOHZ(x)                  (drv_usectohz((x)*1000000))
1139 #define SD_GET_PKT_STATUS(pktp)         ((*(pktp)->pkt_scbp) & STATUS_MASK)

1141 #define SD_BIOERROR(bp, errcode)                                        \
1142         if ((bp)->b_resid == 0) {                                       \
1143                 (bp)->b_resid = (bp)->b_bcount;                         \
1144         }                                                               \
1145         if ((bp)->b_error == 0) {                                       \
1146                 bioerror(bp, errcode);                                  \
1147         }                                                               \
1148         (bp)->b_flags |= B_ERROR;

1150 #define SD_FILL_SCSI1_LUN_CDB(lunp, cdbp)                               \
1151         if (! (lunp)->un_f_is_fibre &&                                  \
1152             SD_INQUIRY((lunp))->inq_ansi == 0x01) {                     \
1153                 int _lun = ddi_prop_get_int(DDI_DEV_T_ANY,              \
1154                     SD_DEVINFO((lunp)), DDI_PROP_DONTPASS,              \
1155                     SCSI_ADDR_PROP_LUN, 0);                             \
1156                 if (_lun > 0) {                                         \
1157                         (cdbp)->scc_lun = _lun;                         \
```



```
1158                 }                                                      \
1159         }

1161 #define SD_FILL_SCSI1_LUN(lunp, pktp)                                   \
1162         SD_FILL_SCSI1_LUN_CDB((lunp), (union scsi_cdb *)(pktp)->pkt_cdbp)

1164 /*
1165  * Disk driver states
1166  */

1168 #define SD_STATE_NORMAL         0
1169 #define SD_STATE_OFFLINE        1
1170 #define SD_STATE_RWAIT          2
1171 #define SD_STATE_DUMPING        3
1172 #define SD_STATE_SUSPENDED      4
1173 #define SD_STATE_PM_CHANGING    5

1175 /*
1176  * The table is to be interpreted as follows: The rows lists all the states
1177  * and each column is a state that a state in each row *can* reach. The entries
1178  * in the table list the event that cause that transition to take place.
1179  * For e.g.: To go from state RWAIT to SUSPENDED, event (d)-- which is the
1180  * invocation of DDI_SUSPEND-- has to take place. Note the same event could
1181  * cause the transition from one state to two different states. e.g., from
1182  * state SUSPENDED, when we get a DDI_RESUME, we just go back to the *last
1183  * state* whatever that might be. (NORMAL or OFFLINE).
1184  *
1185  *
1186  * State Transition Table:
1187  *
1188  *                      NORMAL  OFFLINE  RWAIT  DUMPING  SUSPENDED  PM_SUSPENDED
1189  *
1190  *   NORMAL               -      (a)      (b)     (c)      (d)       (h)
1191  *
1192  *   OFFLINE             (e)      -       (e)     (c)      (d)       NP
1193  *
1194  *   RWAIT               (f)     NP        -      (c)      (d)       (h)
1195  *
1196  *   DUMPING             NP      NP        NP      -        NP       NP
1197  *
1198  *   SUSPENDED           (g)     (g)       (b)     NP*      -        NP
1199  *
1200  *   PM_SUSPENDED        (i)     NP        (b)    (c)      (d)       -
1201  *
1202  *   NP :       Not Possible.
1203  *   (a):       Disk does not respond.
1204  *   (b):       Packet Allocation Fails
1205  *   (c):       Panic - Crash dump
1206  *   (d):       DDI_SUSPEND is called.
1207  *   (e):       Disk has a successful I/O completed.
1208  *   (f):       sdrunout() calls sdstart() which sets it NORMAL
1209  *   (g):       DDI_RESUME is called.
1210  *   (h):       Device threshold exceeded pm framework called power
1211  *              entry point or pm_lower_power called in detach.
1212  *   (i):       When new I/O come in.
1213  *    * :       When suspended, we dont change state during panic dump
1214  */


1217 #define SD_MAX_THROTTLE         256
1218 #define SD_MIN_THROTTLE         8
1219 /*
1220  * Lowest valid max. and min. throttle value.
1221  * This is set to 2 because if un_min_throttle were allowed to be 1 then
1222  * un_throttle would never get set to a value less than un_min_throttle
1223  * (0 is a special case) which means it would never get set back to
```

```
1224  * un_saved_throttle in routine sd_restore_throttle().
1225  */
1226 #define SD_LOWEST_VALID_THROTTLE        2



1230 /* Return codes for sd_send_polled_cmd() and sd_scsi_poll() */
1231 #define SD_CMD_SUCCESS                  0
1232 #define SD_CMD_FAILURE                  1
1233 #define SD_CMD_RESERVATION_CONFLICT     2
1234 #define SD_CMD_ILLEGAL_REQUEST          3
1235 #define SD_CMD_BECOMING_READY           4
1236 #define SD_CMD_CHECK_CONDITION          5

1238 /* Return codes for sd_ready_and_valid */
1239 #define SD_READY_VALID                  0
1240 #define SD_NOT_READY_VALID              1
1241 #define SD_RESERVED_BY_OTHERS           2

1243 #define SD_PATH_STANDARD                0
1244 #define SD_PATH_DIRECT                  1
1245 #define SD_PATH_DIRECT_PRIORITY         2

1247 #define SD_UNIT_ATTENTION_RETRY         40

1249 /*
1250  * The following three are bit flags passed into sd_send_scsi_TEST_UNIT_READY
1251  * to control specific behavior.
1252  */
1253 #define SD_CHECK_FOR_MEDIA              0x01
1254 #define SD_DONT_RETRY_TUR               0x02
1255 #define SD_BYPASS_PM                    0x04

1257 #define SD_GROUP0_MAX_ADDRESS   (0x1fffff)
1258 #define SD_GROUP0_MAXCOUNT      (0xff)
1259 #define SD_GROUP1_MAX_ADDRESS   (0xffffffff)
1260 #define SD_GROUP1_MAXCOUNT      (0xffff)

1262 #define SD_BECOMING_ACTIVE      0x01
1263 #define SD_REMOVAL_ALLOW        0
1264 #define SD_REMOVAL_PREVENT      1

1266 #define SD_GET_PKT_OPCODE(pktp) \
1267         (((union scsi_cdb *)((pktp)->pkt_cdbp))->cdb_un.cmd)


1270 #define SD_NO_RETRY_ISSUED              0
1271 #define SD_DELAYED_RETRY_ISSUED         1
1272 #define SD_IMMEDIATE_RETRY_ISSUED       2

1274 #if defined(__i386) || defined(__amd64)
1275 #define SD_UPDATE_B_RESID(bp, pktp)     \
1276         ((bp)->b_resid += (pktp)->pkt_resid + (SD_GET_XBUF(bp)->xb_dma_resid))
1277 #else
1278 #define SD_UPDATE_B_RESID(bp, pktp)     \
1279         ((bp)->b_resid += (pktp)->pkt_resid)
1280 #endif


1283 #define SD_RETRIES_MASK         0x00FF
1284 #define SD_RETRIES_NOCHECK      0x0000
1285 #define SD_RETRIES_STANDARD     0x0001
1286 #define SD_RETRIES_VICTIM       0x0002
1287 #define SD_RETRIES_BUSY         0x0003
1288 #define SD_RETRIES_UA           0x0004
1289 #define SD_RETRIES_ISOLATE      0x8000
```



```
1290 #define SD_RETRIES_FAILFAST     0x4000

1292 #define SD_UPDATE_RESERVATION_STATUS(un, pktp)                          \
1293 if (((pktp)->pkt_reason == CMD_RESET) ||                                \
1294         ((pktp)->pkt_statistics & (STAT_BUS_RESET | STAT_DEV_RESET))) { \
1295         if (((un)->un_resvd_status & SD_RESERVE) == SD_RESERVE) {       \
1296                 (un)->un_resvd_status |=                                \
1297                     (SD_LOST_RESERVE | SD_WANT_RESERVE);                \
1298         }                                                               \
1299 }

1301 #define SD_SENSE_DATA_IS_VALID          0
1302 #define SD_SENSE_DATA_IS_INVALID        1

1304 /*
1305  * Delay (in seconds) before restoring the "throttle limit" back
1306  * to its maximum value.
1307  * 60 seconds is what we will wait for to reset the
1308  * throttle back to it SD_MAX_THROTTLE for TRAN_BUSY.
1309  * 10 seconds for STATUS_QFULL because QFULL will incrementally
1310  * increase the throttle limit until it reaches max value.
1311  */
1312 #define SD_RESET_THROTTLE_TIMEOUT       60
1313 #define SD_QFULL_THROTTLE_TIMEOUT       10

1315 #define SD_THROTTLE_TRAN_BUSY           0
1316 #define SD_THROTTLE_QFULL               1

1318 #define SD_THROTTLE_RESET_INTERVAL      drv_sectohz(sd_reset_throttle_timeout)
1319 #define SD_THROTTLE_RESET_INTERVAL      \
1320         (sd_reset_throttle_timeout * drv_usectohz(1000000))

1320 #define SD_QFULL_THROTTLE_RESET_INTERVAL        \
1321         drv_sectohz(sd_qfull_throttle_timeout)
1323         (sd_qfull_throttle_timeout * drv_usectohz(1000000))


1324 /*
1325  * xb_pkt_flags defines
1326  * SD_XB_DMA_FREED indicates the scsi_pkt has had its DMA resources freed
1327  * by a call to scsi_dmafree(9F). The resources must be reallocated before
1328  *   before a call to scsi_transport can be made again.
1329  * SD_XB_USCSICMD indicates the scsi request is a uscsi request
1330  * SD_XB_INITPKT_MASK: since this field is also used to store flags for
1331  *   a scsi_init_pkt(9F) call, we need a mask to make sure that we don't
1332  *   pass any unintended bits to scsi_init_pkt(9F) (ugly hack).
1333  */
1334 #define SD_XB_DMA_FREED         0x20000000
1335 #define SD_XB_USCSICMD          0x40000000
1336 #define SD_XB_INITPKT_MASK      (PKT_CONSISTENT | PKT_DMA_PARTIAL)

1338 /*
1339  * Extension for the buf(9s) struct that we receive from a higher
1340  * layer. Located by b_private in the buf(9S).  (The previous contents
1341  * of b_private are saved & restored before calling biodone(9F).)
1342  */
1343 struct sd_xbuf {

1345         struct sd_lun   *xb_un;         /* Ptr to associated sd_lun */
1346         struct scsi_pkt *xb_pktp;       /* Ptr to associated scsi_pkt */

1348         /*
1349          * xb_pktinfo points to any optional data that may be needed
1350          * by the initpkt and/or destroypkt functions.  Typical
1351          * use might be to point to a struct uscsi_cmd.
1352          */
```

```
1353         void    *xb_pktinfo;

1355         /*
1356          * Layer-private data area. This may be used by any layer to store
1357          * layer-specific data on a per-IO basis. Typical usage is for an
1358          * iostart routine to save some info here for later use by its
1359          * partner iodone routine.  This area may be used to hold data or
1360          * a pointer to a data block that is allocated/freed by the layer's
1361          * iostart/iodone routines. Allocation & management policy for the
1362          * layer-private area is defined & implemented by each specific
1363          * layer as required.
1364          *
1365          * IMPORTANT: Since a higher layer may depend on the value in the
1366          * xb_private field, each layer must ensure that it returns the
1367          * buf/xbuf to the next higher layer (via SD_NEXT_IODONE()) with
1368          * the SAME VALUE in xb_private as when the buf/xbuf was first
1369          * received by the layer's iostart routine. Typically this is done
1370          * by the iostart routine saving the contents of xb_private into
1371          * a place in the layer-private data area, and the iodone routine
1372          * restoring the value of xb_private before deallocating the
1373          * layer-private data block and calling SD_NEXT_IODONE(). Of course,
1374          * if a layer never modifies xb_private in a buf/xbuf from a higher
1375          * layer, there will be no need to restore the value.
1376          *
1377          * Note that in the case where a layer _creates_ a buf/xbuf (such as
1378          * by calling sd_shadow_buf_alloc()) to pass to a lower layer, it is
1379          * not necessary to preserve the contents of xb_private as there is
1380          * no higher layer dependency on the value of xb_private. Such a
1381          * buf/xbuf must be deallocated by the layer that allocated it and
1382          * must *NEVER* be passed up to a higher layer.
1383          */
1384         void    *xb_private;    /* Layer-private data block */

1386         /*
1387          * We do not use the b_blkno provided in the buf(9S), as we need to
1388          * make adjustments to it in the driver, but it is not a field that
1389          * the driver owns or is free to modify.
1390          */
1391         daddr_t xb_blkno;               /* Absolute block # on target */
1392         uint64_t xb_ena;                /* ena for a specific SCSI command */

1394         int     xb_chain_iostart;       /* iostart side index */
1395         int     xb_chain_iodone;        /* iodone side index */
1396         int     xb_pkt_flags;           /* Flags for scsi_init_pkt() */
1397         ssize_t xb_dma_resid;
1398         short   xb_retry_count;
1399         short   xb_victim_retry_count;
1400         short   xb_ua_retry_count;      /* unit_attention retry counter */
1401         short   xb_nr_retry_count;      /* not ready retry counter */

1403         /*
1404          * Various status and data used when a RQS command is run on
1405          * the behalf of this command.
1406          */
1407         struct buf      *xb_sense_bp;   /* back ptr to buf, for RQS */
1408         uint_t  xb_sense_state;         /* scsi_pkt state of RQS command */
1409         ssize_t xb_sense_resid;         /* residual of RQS command */
1410         uchar_t xb_sense_status;        /* scsi status byte of RQS command */
1411         uchar_t xb_sense_data[SENSE_LENGTH];    /* sense data from RQS cmd */
1412         /*
1413          * Extra sense larger than SENSE_LENGTH will be allocated
1414          * right after xb_sense_data[SENSE_LENGTH]. Please do not
1415          * add any new field after it.
1416          */
1417 };
```

unchanged_portion_omitted_


```
1700 #endif  /* defined(_KERNEL) || defined(_KMEMUSER) */


1703 /*
1704  * 60 seconds is a *very* reasonable amount of time for most slow CD
1705  * operations.
1706  */
1707 #define SD_IO_TIME                      60

1709 /*
1710  * 2 hours is an excessively reasonable amount of time for format operations.
1711  */
1712 #define SD_FMT_TIME                     (120 * 60)

1714 /*
1715  * 5 seconds is what we'll wait if we get a Busy Status back
1716  */
1717 #define SD_BSY_TIMEOUT                  drv_sectohz(5)
1719 #define SD_BSY_TIMEOUT                  (drv_usectohz(5 * 1000000))

1719 /*
1720  * 100 msec. is what we'll wait if we get Unit Attention.
1721  */
1722 #define SD_UA_RETRY_DELAY               (drv_usectohz((clock_t)100000))

1724 /*
1725  * 100 msec. is what we'll wait for restarted commands.
1726  */
1727 #define SD_RESTART_TIMEOUT              (drv_usectohz((clock_t)100000))

1729 /*
1730  * 10s misaligned I/O warning message interval
1731  */
1732 #define SD_RMW_MSG_PRINT_TIMEOUT        drv_sectohz(10)
1734 #define SD_RMW_MSG_PRINT_TIMEOUT        (drv_usectohz((clock_t)10000000))

1734 /*
1735  * 100 msec. is what we'll wait for certain retries for fibre channel
1736  * targets, 0 msec for parallel SCSI.
1737  */
1738 #if defined(__fibre)
1739 #define SD_RETRY_DELAY                  (drv_usectohz(100000))
1740 #else
1741 #define SD_RETRY_DELAY                  ((clock_t)0)
1742 #endif

1744 /*
1745  * 60 seconds is what we will wait for to reset the
1746  * throttle back to it SD_MAX_THROTTLE.
1747  */
1748 #define SD_RESET_THROTTLE_TIMEOUT       60

1750 /*
1751  * Number of times we'll retry a normal operation.
1752  *
1753  * This includes retries due to transport failure
1754  * (need to distinguish between Target and Transport failure)
1755  *
1756  */
1757 #if defined(__fibre)
1758 #define SD_RETRY_COUNT                  3
1759 #else
1760 #define SD_RETRY_COUNT                  5
1761 #endif
```

```
1763 /*
1764  * Number of times we will retry for unit attention.
1765  */
1766 #define SD_UA_RETRY_COUNT               600

1768 #define SD_VICTIM_RETRY_COUNT(un)       (un->un_victim_retry_count)
1769 #define CD_NOT_READY_RETRY_COUNT(un)    (un->un_retry_count * 2)
1770 #define DISK_NOT_READY_RETRY_COUNT(un)  (un->un_retry_count / 2)


1773 /*
1774  * Maximum number of units we can support
1775  * (controlled by room in minor device byte)
1776  *
1777  * Note: this value is out of date.
1778  */
1779 #define SD_MAXUNIT                      32

1781 /*
1782  * 30 seconds is what we will wait for the IO to finish
1783  * before we fail the DDI_SUSPEND
1784  */
1785 #define SD_WAIT_CMDS_COMPLETE           30

1787 /*
1788  * Prevent/allow media removal flags
1789  */
1790 #define SD_REMOVAL_ALLOW                0
1791 #define SD_REMOVAL_PREVENT              1


1794 /*
1795  * Drive Types (and characteristics)
1796  */
1797 #define VIDMAX                          8
1798 #define PIDMAX                          16


1801 /*
1802  * The following #defines and type definitions for the property
1803  * processing component of the sd driver.
1804  */


1807 /* Miscellaneous Definitions */
1808 #define SD_CONF_VERSION_1               1
1809 #define SD_CONF_NOT_USED                32

1811 /*
1812  * "pm-capable" property values and macros
1813  */
1814 #define SD_PM_CAPABLE_UNDEFINED         -1

1816 #define SD_PM_CAPABLE_IS_UNDEFINED(pm_cap)      \
1817         (pm_cap == SD_PM_CAPABLE_UNDEFINED)

1819 #define SD_PM_CAPABLE_IS_FALSE(pm_cap)  \
1820         ((pm_cap & PM_CAPABLE_PM_MASK) == 0)

1822 #define SD_PM_CAPABLE_IS_TRUE(pm_cap)   \
1823         (!SD_PM_CAPABLE_IS_UNDEFINED(pm_cap) && \
1824             ((pm_cap & PM_CAPABLE_PM_MASK) > 0))

1826 #define SD_PM_CAPABLE_IS_SPC_4(pm_cap)  \
1827         ((pm_cap & PM_CAPABLE_PM_MASK) == PM_CAPABLE_SPC4)
```



```
1829 #define SD_PM_CAP_LOG_SUPPORTED(pm_cap) \
1830         ((pm_cap & PM_CAPABLE_LOG_SUPPORTED) ? TRUE : FALSE)

1832 #define SD_PM_CAP_SMART_LOG(pm_cap)     \
1833         ((pm_cap & PM_CAPABLE_SMART_LOG) ? TRUE : FALSE)

1835 /*
1836  * Property data values used in static configuration table
1837  * These are all based on device characteristics.
1838  * For fibre channel devices, the throttle value is usually
1839  * derived from the devices cmd Q depth divided by the number
1840  * of supported initiators.
1841  */
1842 #define ELITE_THROTTLE_VALUE            10
1843 #define SEAGATE_THROTTLE_VALUE          15
1844 #define IBM_THROTTLE_VALUE              15
1845 #define ST31200N_THROTTLE_VALUE         8
1846 #define FUJITSU_THROTTLE_VALUE          15
1847 #define SYMBIOS_THROTTLE_VALUE          16
1848 #define SYMBIOS_NOTREADY_RETRIES        24
1849 #define LSI_THROTTLE_VALUE              16
1850 #define LSI_NOTREADY_RETRIES            24
1851 #define LSI_OEM_NOTREADY_RETRIES        36
1852 #define PURPLE_THROTTLE_VALUE           64
1853 #define PURPLE_BUSY_RETRIES             60
1854 #define PURPLE_RESET_RETRY_COUNT        36
1855 #define PURPLE_RESERVE_RELEASE_TIME     60
1856 #define SVE_BUSY_RETRIES                60
1857 #define SVE_RESET_RETRY_COUNT           36
1858 #define SVE_RESERVE_RELEASE_TIME        60
1859 #define SVE_THROTTLE_VALUE              10
1860 #define SVE_MIN_THROTTLE_VALUE          2
1861 #define SVE_DISKSORT_DISABLED_FLAG      1
1862 #define MASERATI_DISKSORT_DISABLED_FLAG 1
1863 #define MASERATI_LUN_RESET_ENABLED_FLAG 1
1864 #define PIRUS_THROTTLE_VALUE            64
1865 #define PIRUS_NRR_COUNT                 60
1866 #define PIRUS_BUSY_RETRIES              60
1867 #define PIRUS_RESET_RETRY_COUNT         36
1868 #define PIRUS_MIN_THROTTLE_VALUE        16
1869 #define PIRUS_DISKSORT_DISABLED_FLAG    0
1870 #define PIRUS_LUN_RESET_ENABLED_FLAG    1

1872 /*
1873  * Driver Property Bit Flag definitions
1874  *
1875  * Unfortunately, for historical reasons, the bit-flag definitions are
1876  * different on SPARC, INTEL, & FIBRE platforms.
1877  */

1879 /*
1880  * Bit flag telling driver to set throttle from sd.conf sd-config-list
1881  * and driver table.
1882  *
1883  * The max throttle (q-depth) property implementation is for support of
1884  * fibre channel devices that can drop an i/o request when a queue fills
1885  * up. The number of commands sent to the disk from this driver is
1886  * regulated such that queue overflows are avoided.
1887  */
1888 #define SD_CONF_SET_THROTTLE            0
1889 #define SD_CONF_BSET_THROTTLE           (1 << SD_CONF_SET_THROTTLE)

1891 /*
1892  * Bit flag telling driver to set the controller type from sd.conf
1893  * sd-config-list and driver table.
```

```
1894  */
1895 #if defined(__i386) || defined(__amd64)
1896 #define SD_CONF_SET_CTYPE               1
1897 #elif defined(__fibre)
1898 #define SD_CONF_SET_CTYPE               5
1899 #else
1900 #define SD_CONF_SET_CTYPE               1
1901 #endif
1902 #define SD_CONF_BSET_CTYPE              (1 << SD_CONF_SET_CTYPE)

1904 /*
1905  * Bit flag telling driver to set the not ready retry count for a device from
1906  * sd.conf sd-config-list and driver table.
1907  */
1908 #if defined(__i386) || defined(__amd64)
1909 #define SD_CONF_SET_NOTREADY_RETRIES    10
1910 #elif defined(__fibre)
1911 #define SD_CONF_SET_NOTREADY_RETRIES    1
1912 #else
1913 #define SD_CONF_SET_NOTREADY_RETRIES    2
1914 #endif
1915 #define SD_CONF_BSET_NRR_COUNT          (1 << SD_CONF_SET_NOTREADY_RETRIES)

1917 /*
1918  * Bit flag telling driver to set SCSI status BUSY Retries from sd.conf
1919  * sd-config-list and driver table.
1920  */
1921 #if defined(__i386) || defined(__amd64)
1922 #define SD_CONF_SET_BUSY_RETRIES        11
1923 #elif defined(__fibre)
1924 #define SD_CONF_SET_BUSY_RETRIES        2
1925 #else
1926 #define SD_CONF_SET_BUSY_RETRIES        5
1927 #endif
1928 #define SD_CONF_BSET_BSY_RETRY_COUNT    (1 << SD_CONF_SET_BUSY_RETRIES)

1930 /*
1931  * Bit flag telling driver that device does not have a valid/unique serial
1932  * number.
1933  */
1934 #if defined(__i386) || defined(__amd64)
1935 #define SD_CONF_SET_FAB_DEVID           2
1936 #else
1937 #define SD_CONF_SET_FAB_DEVID           3
1938 #endif
1939 #define SD_CONF_BSET_FAB_DEVID          (1 << SD_CONF_SET_FAB_DEVID)

1941 /*
1942  * Bit flag telling driver to disable all caching for disk device.
1943  */
1944 #if defined(__i386) || defined(__amd64)
1945 #define SD_CONF_SET_NOCACHE             3
1946 #else
1947 #define SD_CONF_SET_NOCACHE             4
1948 #endif
1949 #define SD_CONF_BSET_NOCACHE            (1 << SD_CONF_SET_NOCACHE)

1951 /*
1952  * Bit flag telling driver that the PLAY AUDIO command requires parms in BCD
1953  * format rather than binary.
1954  */
1955 #if defined(__i386) || defined(__amd64)
1956 #define SD_CONF_SET_PLAYMSF_BCD         4
1957 #else
1958 #define SD_CONF_SET_PLAYMSF_BCD         6
1959 #endif
```



```
1960 #define SD_CONF_BSET_PLAYMSF_BCD        (1 << SD_CONF_SET_PLAYMSF_BCD)

1962 /*
1963  * Bit flag telling driver that the response from the READ SUBCHANNEL command
1964  * has BCD fields rather than binary.
1965  */
1966 #if defined(__i386) || defined(__amd64)
1967 #define SD_CONF_SET_READSUB_BCD         5
1968 #else
1969 #define SD_CONF_SET_READSUB_BCD         7
1970 #endif
1971 #define SD_CONF_BSET_READSUB_BCD        (1 << SD_CONF_SET_READSUB_BCD)

1973 /*
1974  * Bit in flags telling driver that the track number fields in the READ TOC
1975  * request and respone are in BCD rather than binary.
1976  */
1977 #if defined(__i386) || defined(__amd64)
1978 #define SD_CONF_SET_READ_TOC_TRK_BCD    6
1979 #else
1980 #define SD_CONF_SET_READ_TOC_TRK_BCD    8
1981 #endif
1982 #define SD_CONF_BSET_READ_TOC_TRK_BCD   (1 << SD_CONF_SET_READ_TOC_TRK_BCD)

1984 /*
1985  * Bit flag telling driver that the address fields in the READ TOC request and
1986  * respone are in BCD rather than binary.
1987  */
1988 #if defined(__i386) || defined(__amd64)
1989 #define SD_CONF_SET_READ_TOC_ADDR_BCD   7
1990 #else
1991 #define SD_CONF_SET_READ_TOC_ADDR_BCD   9
1992 #endif
1993 #define SD_CONF_BSET_READ_TOC_ADDR_BCD  (1 << SD_CONF_SET_READ_TOC_ADDR_BCD)

1995 /*
1996  * Bit flag telling the driver that the device doesn't support the READ HEADER
1997  * command.
1998  */
1999 #if defined(__i386) || defined(__amd64)
2000 #define SD_CONF_SET_NO_READ_HEADER      8
2001 #else
2002 #define SD_CONF_SET_NO_READ_HEADER      10
2003 #endif
2004 #define SD_CONF_BSET_NO_READ_HEADER     (1 << SD_CONF_SET_NO_READ_HEADER)

2006 /*
2007  * Bit flag telling the driver that for the READ CD command the device uses
2008  * opcode 0xd4 rather than 0xbe.
2009  */
2010 #if defined(__i386) || defined(__amd64)
2011 #define SD_CONF_SET_READ_CD_XD4         9
2012 #else
2013 #define SD_CONF_SET_READ_CD_XD4         11
2014 #endif
2015 #define SD_CONF_BSET_READ_CD_XD4        (1 << SD_CONF_SET_READ_CD_XD4)

2017 /*
2018  * Bit flag telling the driver to set SCSI status Reset Retries
2019  * (un_reset_retry_count) from sd.conf sd-config-list and driver table (4356701)
2020  */
2021 #define SD_CONF_SET_RST_RETRIES         12
2022 #define SD_CONF_BSET_RST_RETRIES        (1 << SD_CONF_SET_RST_RETRIES)

2024 /*
2025  * Bit flag telling the driver to set the reservation release timeout value
```

```
2026  * from sd.conf sd-config-list and driver table. (4367306)
2027  */
2028 #define SD_CONF_SET_RSV_REL_TIME        13
2029 #define SD_CONF_BSET_RSV_REL_TIME       (1 << SD_CONF_SET_RSV_REL_TIME)

2031 /*
2032  * Bit flag telling the driver to verify that no commands are pending for a
2033  * device before issuing a Test Unit Ready. This is a fw workaround for Seagate
2034  * eliteI drives. (4392016)
2035  */
2036 #define SD_CONF_SET_TUR_CHECK           14
2037 #define SD_CONF_BSET_TUR_CHECK          (1 << SD_CONF_SET_TUR_CHECK)

2039 /*
2040  * Bit in flags telling driver to set min. throttle from ssd.conf
2041  * ssd-config-list and driver table.
2042  */
2043 #define SD_CONF_SET_MIN_THROTTLE        15
2044 #define SD_CONF_BSET_MIN_THROTTLE       (1 << SD_CONF_SET_MIN_THROTTLE)

2046 /*
2047  * Bit in flags telling driver to set disksort disable flag from ssd.conf
2048  * ssd-config-list and driver table.
2049  */
2050 #define SD_CONF_SET_DISKSORT_DISABLED   16
2051 #define SD_CONF_BSET_DISKSORT_DISABLED  (1 << SD_CONF_SET_DISKSORT_DISABLED)

2053 /*
2054  * Bit in flags telling driver to set LUN Reset enable flag from [s]sd.conf
2055  * [s]sd-config-list and driver table.
2056  */
2057 #define SD_CONF_SET_LUN_RESET_ENABLED   17
2058 #define SD_CONF_BSET_LUN_RESET_ENABLED  (1 << SD_CONF_SET_LUN_RESET_ENABLED)

2060 /*
2061  * Bit in flags telling driver that the write cache on the device is
2062  * non-volatile.
2063  */
2064 #define SD_CONF_SET_CACHE_IS_NV 18
2065 #define SD_CONF_BSET_CACHE_IS_NV        (1 << SD_CONF_SET_CACHE_IS_NV)

2067 /*
2068  * Bit in flags telling driver that the power condition flag from [s]sd.conf
2069  * [s]sd-config-list and driver table.
2070  */
2071 #define SD_CONF_SET_PC_DISABLED 19
2072 #define SD_CONF_BSET_PC_DISABLED        (1 << SD_CONF_SET_PC_DISABLED)

2074 /*
2075  * This is the number of items currently settable in the sd.conf
2076  * sd-config-list.  The mask value is defined for parameter checking. The
2077  * item count and mask should be updated when new properties are added.
2078  */
2079 #define SD_CONF_MAX_ITEMS               19
2080 #define SD_CONF_BIT_MASK                0x0007FFFF

2082 typedef struct {
2083         int sdt_throttle;
2084         int sdt_ctype;
2085         int sdt_not_rdy_retries;
2086         int sdt_busy_retries;
2087         int sdt_reset_retries;
2088         int sdt_reserv_rel_time;
2089         int sdt_min_throttle;
2090         int sdt_disk_sort_dis;
2091         int sdt_lun_reset_enable;
```



```
2092         int sdt_suppress_cache_flush;
2093         int sdt_power_condition_dis;
2094 } sd_tunables;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5911 Wed Aug 19 07:25:24 2015
new/usr/src/uts/common/sys/scsi/targets/sgendef.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 154 #define SGEN_ESTIMATED_NUM_DEVS 4               /* for soft-state allocation */

 156 /*
 157  * Time to wait before a retry for commands returning Busy Status
 158  */
 159 #define SGEN_BSY_TIMEOUT        drv_sectohz(5)
 159 #define SGEN_BSY_TIMEOUT        (drv_usectohz(5 * 1000000))
 160 #define SGEN_IO_TIME            60              /* seconds */

 162 /*
 163  * sgen_callback action codes
 164  */
 165 #define COMMAND_DONE            0       /* command completed, biodone it */
 166 #define COMMAND_DONE_ERROR      1       /* command completed, indicate error */
 167 #define FETCH_SENSE             2       /* CHECK CONDITION, so initiate sense */
 168                                         /* fetch */

 170 #define SET_BP_ERROR(bp, err)   bioerror(bp, err);

 172 #endif /* defined(_KERNEL) */

 174 #ifdef  __cplusplus
 175 }
______unchanged_portion_omitted_
```

```
**********************************************************
   13424 Wed Aug 19 07:25:24 2015
new/usr/src/uts/common/sys/stmf.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 246 /*
 247  * conditions causing or affecting the change.
 248  */
 249 #define STMF_RFLAG_USER_REQUEST         0x0001
 250 #define STMF_RFLAG_FATAL_ERROR          0x0002
 251 #define STMF_RFLAG_STAY_OFFLINED        0x0004
 252 #define STMF_RFLAG_RESET                0x0008
 253 #define STMF_RFLAG_COLLECT_DEBUG_DUMP   0x0010
 254 #define STMF_RFLAG_LU_ABORT             0x0020
 255 #define STMF_RFLAG_LPORT_ABORT          0x0040

 257 #define STMF_CHANGE_INFO_LEN            160

 259 /*
 260  * cmds to stmf_abort entry point
 261  */
 262 #define STMF_QUEUE_TASK_ABORT           1
 263 #define STMF_REQUEUE_TASK_ABORT_LPORT   2
 264 #define STMF_REQUEUE_TASK_ABORT_LU      3
 265 #define STMF_QUEUE_ABORT_LU             4

 267 /*
 268  * cmds to be used by stmf ctl
 269  */
 270 #define STMF_CMD_LU_OP                  0x0100
 271 #define STMF_CMD_LPORT_OP               0x0200
 272 #define STMF_CMD_MASK                   0x00ff
 273 #define STMF_CMD_ONLINE                 0x0001
 274 #define STMF_CMD_OFFLINE                0x0002
 275 #define STMF_CMD_GET_STATUS             0x0003
 276 #define STMF_CMD_ONLINE_COMPLETE        0x0004
 277 #define STMF_CMD_OFFLINE_COMPLETE       0x0005
 278 #define STMF_ACK_ONLINE_COMPLETE        0x0006
 279 #define STMF_ACK_OFFLINE_COMPLETE       0x0007

 281 #define STMF_CMD_LU_ONLINE              (STMF_CMD_LU_OP | STMF_CMD_ONLINE)
 282 #define STMF_CMD_LU_OFFLINE             (STMF_CMD_LU_OP | STMF_CMD_OFFLINE)
 283 #define STMF_CMD_LPORT_ONLINE           (STMF_CMD_LPORT_OP | STMF_CMD_ONLINE)
 284 #define STMF_CMD_LPORT_OFFLINE          (STMF_CMD_LPORT_OP | STMF_CMD_OFFLINE)
 285 #define STMF_CMD_GET_LU_STATUS          (STMF_CMD_LU_OP | STMF_CMD_GET_STATUS)
 286 #define STMF_CMD_GET_LPORT_STATUS       \
 287                         (STMF_CMD_LPORT_OP | STMF_CMD_GET_STATUS)
 288 #define STMF_CMD_LU_ONLINE_COMPLETE     \
 289                         (STMF_CMD_LU_OP | STMF_CMD_ONLINE_COMPLETE)
 290 #define STMF_CMD_LPORT_ONLINE_COMPLETE  \
 291                         (STMF_CMD_LPORT_OP | STMF_CMD_ONLINE_COMPLETE)
 292 #define STMF_ACK_LU_ONLINE_COMPLETE     \
 293                         (STMF_CMD_LU_OP | STMF_ACK_ONLINE_COMPLETE)
 294 #define STMF_ACK_LPORT_ONLINE_COMPLETE  \
 295                         (STMF_CMD_LPORT_OP | STMF_ACK_ONLINE_COMPLETE)
 296 #define STMF_CMD_LU_OFFLINE_COMPLETE    \
 297                         (STMF_CMD_LU_OP | STMF_CMD_OFFLINE_COMPLETE)
 298 #define STMF_CMD_LPORT_OFFLINE_COMPLETE \
 299                         (STMF_CMD_LPORT_OP | STMF_CMD_OFFLINE_COMPLETE)
 300 #define STMF_ACK_LU_OFFLINE_COMPLETE    \
 301                         (STMF_CMD_LU_OP | STMF_ACK_OFFLINE_COMPLETE)
 302 #define STMF_ACK_LPORT_OFFLINE_COMPLETE \
 303                         (STMF_CMD_LPORT_OP | STMF_ACK_OFFLINE_COMPLETE)
 304 /*
```



```
 305  * For LPORTs and LUs to create their own ctl cmds which dont
 306  * conflict with stmf ctl cmds.
 307  */
 308 #define STMF_LPORT_CTL_CMDS             0x1000
 309 #define STMF_LU_CTL_CMDS                0x2000

 311 /*
 312  * Commands for various info routines.
 313  */
 314 /* Command classifiers */
 315 #define SI_LPORT                0x1000000
 316 #define SI_STMF                 0x2000000
 317 #define SI_LU                   0x4000000
 318 #define SI_LPORT_FC             0x0000000
 319 #define SI_LPORT_ISCSI          0x0010000
 320 #define SI_LPORT_SAS            0x0020000
 321 #define SI_STMF_LU              0x0010000
 322 #define SI_STMF_LPORT           0x0020000

 324 #define SI_GET_CLASS(v)         ((v) & 0xFF000000)
 325 #define SI_GET_SUBCLASS(v)      ((v) & 0x00FF0000)

 327 /* Commands for LPORT info routines */
 328 /* XXX - Implement these. */
 329 #if 0
 330 #define SI_LPORT_FC_PORTINFO            (SI_LPORT | SI_LPORT_FC | 1)
 331 #define SI_RPORT_FC_PORTINFO            (SI_LPORT | SI_LPORT_FC | 2)
 332 #endif

 334 /*
 335  * Events
 336  */
 337 #define STMF_EVENT_ALL                  ((int)-1)
 338 #define LPORT_EVENT_INITIAL_LUN_MAPPED  0

 340 /*
 341  * This needs to go into common/ddi/sunddi.h
 342  */
 343 #define DDI_NT_STMF             "ddi_scsi_target:framework"
 344 #define DDI_NT_STMF_LP          "ddi_scsi_target:lu_provider"
 345 #define DDI_NT_STMF_PP          "ddi_scsi_target:port_provider"

 347 /*
 348  * VPD page bits.
 349  */
 350 #define STMF_VPD_LU_ID          0x01
 351 #define STMF_VPD_TARGET_ID      0x02
 352 #define STMF_VPD_TP_GROUP       0x04
 353 #define STMF_VPD_RELATIVE_TP_ID 0x08

 355 /*
 356  * Common macros to simplify coding
 357  */
 358 #define STMF_SEC2TICK(x_sec)    drv_sectohz(x_sec)
 358 #define STMF_SEC2TICK(x_sec)    (drv_usectohz((x_sec) * 1000000))

 360 void stmf_trace(caddr_t ident, const char *fmt, ...);
 361 void *stmf_alloc(stmf_struct_id_t sid, int additional_size, int alloc_flags);
 362 void stmf_free(void *struct_ptr);
 363 struct scsi_task *stmf_task_alloc(struct stmf_local_port *lport,
 364     struct stmf_scsi_session *ss, uint8_t *lun, uint16_t cdb_length,
 365     uint16_t ext_id);
 366 void stmf_post_task(scsi_task_t *task, stmf_data_buf_t *dbuf);
 367 stmf_data_buf_t *stmf_alloc_dbuf(scsi_task_t *task, uint32_t size,
 368     uint32_t *pminsize, uint32_t flags);
 369 void stmf_free_dbuf(scsi_task_t *task, stmf_data_buf_t *dbuf);
```

```
370 stmf_status_t stmf_setup_dbuf(scsi_task_t *task, stmf_data_buf_t *dbuf,
371     uint32_t flags);
372 void stmf_teardown_dbuf(scsi_task_t *task, stmf_data_buf_t *dbuf);
373 stmf_status_t stmf_xfer_data(scsi_task_t *task, stmf_data_buf_t *dbuf,
374     uint32_t ioflags);
375 stmf_status_t stmf_send_scsi_status(scsi_task_t *task, uint32_t ioflags);
376 void stmf_data_xfer_done(scsi_task_t *task, stmf_data_buf_t *dbuf,
377     uint32_t iof);
378 void stmf_send_status_done(scsi_task_t *task, stmf_status_t s, uint32_t iof);
379 void stmf_task_lu_done(scsi_task_t *task);
380 void stmf_abort(int abort_cmd, scsi_task_t *task, stmf_status_t s, void *arg);
381 void stmf_task_lu_aborted(scsi_task_t *task, stmf_status_t s, uint32_t iof);
382 void stmf_task_lport_aborted(scsi_task_t *task, stmf_status_t s, uint32_t iof);
383 stmf_status_t stmf_task_poll_lu(scsi_task_t *task, uint32_t timeout);
384 stmf_status_t stmf_task_poll_lport(scsi_task_t *task, uint32_t timeout);
385 stmf_status_t stmf_ctl(int cmd, void *obj, void *arg);
386 stmf_status_t stmf_register_itl_handle(struct stmf_lu *lu, uint8_t *lun,
387     struct stmf_scsi_session *ss, uint64_t session_id, void *itl_handle);
388 stmf_status_t stmf_deregister_all_lu_itl_handles(struct stmf_lu *lu);
389 stmf_status_t stmf_get_itl_handle(struct stmf_lu *lu, uint8_t *lun,
390     struct stmf_scsi_session *ss, uint64_t session_id, void **itl_handle_retp);
391 stmf_data_buf_t *stmf_handle_to_buf(scsi_task_t *task, uint8_t h);
392 stmf_status_t stmf_lu_add_event(struct stmf_lu *lu, int eventid);
393 stmf_status_t stmf_lu_remove_event(struct stmf_lu *lu, int eventid);
394 stmf_status_t stmf_lport_add_event(struct stmf_local_port *lport, int eventid);
395 stmf_status_t stmf_lport_remove_event(struct stmf_local_port *lport,
396     int eventid);
397 void stmf_wwn_to_devid_desc(struct scsi_devid_desc *sdid, uint8_t *wwn,
398     uint8_t protocol_id);
399 stmf_status_t stmf_scsilib_uniq_lu_id(uint32_t company_id,
400     struct scsi_devid_desc *lu_id);
401 stmf_status_t stmf_scsilib_uniq_lu_id2(uint32_t company_id, uint32_t host_id,
402     struct scsi_devid_desc *lu_id);
403 void stmf_scsilib_send_status(scsi_task_t *task, uint8_t st, uint32_t saa);
404 uint32_t stmf_scsilib_prepare_vpd_page83(scsi_task_t *task, uint8_t *page,
405             uint32_t page_len, uint8_t byte0, uint32_t vpd_mask);
406 uint16_t stmf_scsilib_get_lport_rtid(struct scsi_devid_desc *devid);
407 struct scsi_devid_desc *stmf_scsilib_get_devid_desc(uint16_t rtpid);
408 void stmf_scsilib_handle_report_tpgs(scsi_task_t *task, stmf_data_buf_t *dbuf);
409 void stmf_scsilib_handle_task_mgmt(scsi_task_t *task);

411 struct stmf_remote_port *stmf_scsilib_devid_to_remote_port(
412     struct scsi_devid_desc *);
413 boolean_t stmf_scsilib_tptid_validate(struct scsi_transport_id *,
414     uint32_t, uint16_t *);
415 boolean_t stmf_scsilib_tptid_compare(struct scsi_transport_id *,
416     struct scsi_transport_id *);
417 struct stmf_remote_port *stmf_remote_port_alloc(uint16_t);
418 void stmf_remote_port_free(struct stmf_remote_port *);
419 #ifdef  __cplusplus
420 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10239 Wed Aug 19 07:25:24 2015
new/usr/src/uts/common/sys/usb/clients/audio/usb_ac/usb_ac.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 266 /* warlock directives, stable data */
 267 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_state_t))
 268 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_power_t))
 269 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_plumbed_t))
 270 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_eng_t))
 271 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_format_t))
 272 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_ctrl_t::ctrl_mutex, usb_audio_ctrl_t))


 275 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dip))
 276 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ser_acc))
 277 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_pm))
 278 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_instance))
 279 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_default_ph))
 280 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_log_handle))
 281 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_if_descr))
 282 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dev_data))
 283 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ifno))
 284 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::flags))
 285 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_input_ports))
 286 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::engines))
 287 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_audio_dev))
 288 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::controls))

 290 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_eflags))
 291 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::streams))
 292 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::statep))
 293 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fmt))
 294 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fragfr))
 295 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::frsmshift))
 296 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::started))
 297 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_engp))
 298 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::io_count))
 299 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::intrate))

 301 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::statep))
 302 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::af_ctrlp))
 303 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::cval))

 305 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_tqp))
 306 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_uacp))

 308 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_format_t::ch))

 310 /* usb_ac driver only care about two states:  plumbed or unplumbed */
 311 #define USB_AC_STATE_UNPLUMBED          0
 312 #define USB_AC_STATE_PLUMBED            1
 313 #define USB_AC_STATE_PLUMBED_RESTORING  2

 315 /* Default pipe states */
 316 #define USB_AC_DEF_CLOSED               0
 317 #define USB_AC_DEF_OPENED               1

 319 #define USB_AC_BUFFER_SIZE              256     /* descriptor buffer size */


 322 /*
 323  * delay before restoring state
 324  */
```



```
 325 #define USB_AC_RESTORE_DELAY            drv_sectohz(1)
 325 #define USB_AC_RESTORE_DELAY            drv_usectohz(1000000)

 327 /* value for acp_driver */
 328 #define USB_AS_PLUMBED  1
 329 #define USB_AH_PLUMBED  2
 330 #define UNKNOWN_PLUMBED 3

 332 /* other useful macros */
 333 #define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))




 340 #define AF_REGISTERED   0x1
 341 #define AD_SETUP        0x10


 344 int usb_audio_attach(usb_ac_state_t *);
 345 /*
 346  * framework gain range
 347  */
 348 #define AUDIO_CTRL_STEREO_VAL(l, r)     (((l) & 0xff) | (((r) & 0xff) << 8))
 349 #define AUDIO_CTRL_STEREO_LEFT(v)       ((uint8_t)((v) & 0xff))
 350 #define AUDIO_CTRL_STEREO_RIGHT(v)      ((uint8_t)(((v) >> 8) & 0xff))


 353 #define AF_MAX_GAIN     100
 354 #define AF_MIN_GAIN     0



 358 int usb_ac_get_audio(void *, void *, int);

 360 void usb_ac_send_audio(void *, void *, int);

 362 void usb_ac_stop_play(usb_ac_state_t *, usb_audio_eng_t *);


 365 #ifdef __cplusplus
 366 }
______unchanged_portion_omitted_
```

```
**********************************************************
   19897 Wed Aug 19 07:25:24 2015
new/usr/src/uts/common/sys/usb/hcd/uhci/uhcid.h
XXXX introduce drv_sectohz
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #ifndef _SYS_USB_UHCID_H
  27 #define _SYS_USB_UHCID_H


  30 #ifdef __cplusplus
  31 extern "C" {
  32 #endif

  34 /*
  35  * Universal Host Controller Driver (UHCI)
  36  *
  37  * The UHCI driver is a driver which interfaces to the Universal
  38  * Serial Bus Driver (USBA) and the Host Controller (HC). The interface to
  39  * the Host Controller is defined by the Universal Host Controller Interface.
  40  *
  41  * This file contains the data structures for the UHCI driver.
  42  */
  43 #include <sys/types.h>
  44 #include <sys/pci.h>
  45 #include <sys/kstat.h>

  47 #include <sys/usb/usba/usbai_version.h>
  48 #include <sys/usb/usba.h>
  49 #include <sys/usb/usba/usba_types.h>

  51 #include <sys/usb/usba/genconsole.h>
  52 #include <sys/usb/usba/hcdi.h>

  54 #include <sys/usb/hubd/hub.h>
  55 #include <sys/usb/usba/hubdi.h>
  56 #include <sys/usb/hubd/hubdvar.h>

  58 #include <sys/usb/hcd/uhci/uhci.h>

  60 /* limit the xfer size for bulk */
  61 #define UHCI_BULK_MAX_XFER_SIZE (124*1024) /* Max bulk xfer size */
```



```
  63 /* Maximum allowable data transfer size per transaction */
  64 #define UHCI_MAX_TD_XFER_SIZE   0x500 /* Maximum data per transaction */

  66 /*
  67  * Generic UHCI Macro definitions
  68  */
  69 #define UHCI_UNDERRUN_OCCURRED  0x1234
  70 #define UHCI_OVERRUN_OCCURRED   0x5678
  71 #define UHCI_PROP_MASK          0x01000020
  72 #define UHCI_RESET_DELAY        15000
  73 #define UHCI_TIMEWAIT           10000

  75 #define MAX_SOF_WAIT_COUNT      2
  76 #define MAX_RH_PORTS            2
  77 #define DISCONNECTED            2
  78 #define POLLING_FREQ_7MS        7
  79 #define PCI_CONF_IOBASE         0x20
  80 #define PCI_CONF_IOBASE_MASK    0xffe0

  82 #define UHCI_ONE_SECOND         drv_sectohz(1)
  82 #define UHCI_ONE_SECOND         drv_usectohz(1000000)
  83 #define UHCI_ONE_MS             drv_usectohz(1000)
  84 #define UHCI_32_MS              drv_usectohz(32*1000)
  85 #define UHCI_256_MS             drv_usectohz(256*1000)
  86 #define UHCI_MAX_INSTS          4

  88 #define POLLED_RAW_BUF_SIZE     8

  90 /* Default time out values for bulk and ctrl commands */
  91 #define UHCI_CTRL_TIMEOUT       5
  92 #define UHCI_BULK_TIMEOUT       60

  94 /* UHCI root hub structure */
  95 typedef struct uhci_root_hub_info {
  96         uint_t                  rh_status;              /* Last RH status */
  97         uint_t                  rh_num_ports;           /* #ports on the root */

  99         /* Last status of ports */
 100         uint_t                  rh_port_status[MAX_RH_PORTS];
 101         uint_t                  rh_port_changes[MAX_RH_PORTS];
 102         uint_t                  rh_port_state[MAX_RH_PORTS]; /* See below */

 104         usba_pipe_handle_data_t *rh_intr_pipe_handle;   /* RH intr pipe hndle */
 105         usb_hub_descr_t         rh_descr;               /* RH descr's copy */
 106         uint_t                  rh_pipe_state;          /* RH intr pipe state */

 108         usb_intr_req_t          *rh_curr_intr_reqp;     /* Current intr req */
 109         usb_intr_req_t          *rh_client_intr_req;    /* save IN request */
 110 } uhci_root_hub_info_t;
______unchanged_portion_omitted_
```

****
18234 Wed Aug 19 07:25:24 2015
new/usr/src/uts/i86pc/i86hvm/io/xpv/xpv_support.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 560 static void
 561 xen_shutdown(void *arg)
 562 {
 563         int cmd = (uintptr_t)arg;
 564         proc_t *initpp;

 566         ASSERT(cmd > SHUTDOWN_INVALID && cmd < SHUTDOWN_MAX);

 568         if (cmd == SHUTDOWN_SUSPEND) {
 569                 xen_suspend_domain();
 570                 return;
 571         }

 573         switch (cmd) {
 574         case SHUTDOWN_POWEROFF:
 575                 force_shutdown_method = AD_POWEROFF;
 576                 break;
 577         case SHUTDOWN_HALT:
 578                 force_shutdown_method = AD_HALT;
 579                 break;
 580         case SHUTDOWN_REBOOT:
 581                 force_shutdown_method = AD_BOOT;
 582                 break;
 583         }


 586         /*
 587          * If we're still booting and init(1) isn't set up yet, simply halt.
 588          */
 589         mutex_enter(&pidlock);
 590         initpp = prfind(P_INITPID);
 591         mutex_exit(&pidlock);
 592         if (initpp == NULL) {
 593                 extern void halt(char *);
 594                 halt("Power off the System");   /* just in case */
 595         }

 597         /*
 598          * else, graceful shutdown with inittab and all getting involved
 599          */
 600         psignal(initpp, SIGPWR);

 602         (void) timeout(xen_dirty_shutdown, arg,
 603             drv_sectohz(SHUTDOWN_TIMEOUT_SECS));
 603             SHUTDOWN_TIMEOUT_SECS * drv_usectohz(MICROSEC));
 604 }
```

_____***unchanged_portion_omitted_***

****
17850 Wed Aug 19 07:25:24 2015
new/usr/src/uts/i86pc/io/tzmon/tzmon.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
573 /*
574  * tzmon_monitor
575  * Run as a separate thread, this wakes according to polling period and
576  * checks particular objects in the thermal zone.  One instance per
577  * thermal zone.
578  */
579 static void
580 tzmon_monitor(void *ctx)
581 {
582         thermal_zone_t *tzp = (thermal_zone_t *)ctx;
583         clock_t ticks;

585         do {
586                 /* Check out the zone */
587                 tzmon_eval_zone(tzp);

589                 /* Go back to sleep */
590                 mutex_enter(&tzp->lock);
591                 ticks = drv_sectohz(tzp->polling_period);
591                 ticks = drv_usectohz(tzp->polling_period * 1000000);
592                 if (ticks > 0)
593                         (void) cv_reltimedwait(&zone_list_condvar,
594                             &tzp->lock, ticks, TR_CLOCK_TICK);
595                 mutex_exit(&tzp->lock);
596         } while (ticks > 0);
597 }
```

_____***unchanged_portion_omitted_***

****
5941 Wed Aug 19 07:25:25 2015
new/usr/src/uts/i86xpv/cpu/generic_cpu/gcpu_poll_xpv.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 112 static void
 113 gcpu_xpv_mch_poll(void *arg)
 114 {
 115         cmi_hdl_t hdl = cmi_hdl_any();

 117         if (hdl != NULL) {
 118                 cmi_mc_logout(hdl, 0, 0);
 119                 cmi_hdl_rele(hdl);
 120         }

 122         if (arg == GCPU_XPV_MCH_POLL_REARM &&
 123             gcpu_xpv_mch_poll_interval_secs != 0) {
 124                 gcpu_xpv_mch_poll_timeoutid = timeout(gcpu_xpv_mch_poll,
 125                     GCPU_XPV_MCH_POLL_REARM,
 126                     drv_sectohz(gcpu_xpv_mch_poll_interval_secs));
 126                     drv_usectohz(gcpu_xpv_mch_poll_interval_secs * MICROSEC));
 127         }
 128 }
```

_____***unchanged_portion_omitted_***

```
 170 void
 171 gcpu_mca_poll_start(cmi_hdl_t hdl)
 172 {
 173         ASSERT(cmi_hdl_class(hdl) == CMI_HDL_SOLARIS_xVM_MCA);
 174         /*
 175          * We are on the boot cpu (cpu 0), called at the end of its
 176          * multiprocessor startup.
 177          */
 178         if (gcpu_xpv_poll_bankregs_sz != 0 && gcpu_xpv_virq_vect == -1) {
 179                 /*
 180                  * The hypervisor will poll MCA state for us, but it cannot
 181                  * poll MCH state so we do that via a timeout.
 182                  */
 183                 if (gcpu_xpv_mch_poll_interval_secs != 0) {
 184                         gcpu_xpv_mch_poll_timeoutid =
 185                             timeout(gcpu_xpv_mch_poll, GCPU_XPV_MCH_POLL_REARM,
 186                             drv_sectohz(gcpu_xpv_mch_poll_interval_secs));
 186                             drv_usectohz(gcpu_xpv_mch_poll_interval_secs *
 187                             MICROSEC));
 187                 }

 189                 /*
 190                  * Register handler for VIRQ_MCA; once this is in place
 191                  * the hypervisor will begin to forward polled MCA observations
 192                  * to us.
 193                  */
 194                 gcpu_xpv_virq_vect = ec_bind_virq_to_irq(VIRQ_MCA, 0);
 195                 (void) add_avintr(NULL, gcpu_xpv_virq_level,
 196                     (avfunc)gcpu_xpv_virq_intr, "MCA", gcpu_xpv_virq_vect,
 197                     NULL, NULL, NULL, NULL);
 198         }
 199 }
```

_____***unchanged_portion_omitted_***

****
32960 Wed Aug 19 07:25:25 2015
new/usr/src/uts/i86xpv/os/xen_machdep.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 761 static void
 762 xen_shutdown(void *arg)
 763 {
 764         int cmd = (uintptr_t)arg;
 765         proc_t *initpp;

 767         ASSERT(cmd > SHUTDOWN_INVALID && cmd < SHUTDOWN_MAX);

 769         if (cmd == SHUTDOWN_SUSPEND) {
 770                 xen_suspend_domain();
 771                 return;
 772         }

 774         switch (cmd) {
 775         case SHUTDOWN_POWEROFF:
 776                 force_shutdown_method = AD_POWEROFF;
 777                 break;
 778         case SHUTDOWN_HALT:
 779                 force_shutdown_method = AD_HALT;
 780                 break;
 781         case SHUTDOWN_REBOOT:
 782                 force_shutdown_method = AD_BOOT;
 783                 break;
 784         }

 786         /*
 787          * If we're still booting and init(1) isn't set up yet, simply halt.
 788          */
 789         mutex_enter(&pidlock);
 790         initpp = prfind(P_INITPID);
 791         mutex_exit(&pidlock);
 792         if (initpp == NULL) {
 793                 extern void halt(char *);
 794                 halt("Power off the System");   /* just in case */
 795         }

 797         /*
 798          * else, graceful shutdown with inittab and all getting involved
 799          */
 800         psignal(initpp, SIGPWR);

 802         (void) timeout(xen_dirty_shutdown, arg,
 803             drv_sectohz(SHUTDOWN_TIMEOUT_SECS));
 803             SHUTDOWN_TIMEOUT_SECS * drv_usectohz(MICROSEC));
 804 }
```

_____***unchanged_portion_omitted_***

****
51883 Wed Aug 19 07:25:25 2015
new/usr/src/uts/intel/io/dktp/disk/cmdk.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 610 /*
 611  * suspend routine, it will be run when get the command
 612  * DDI_SUSPEND at detach(9E) from system power management
 613  */
 614 static int
 615 cmdksuspend(dev_info_t *dip)
 616 {
 617         struct cmdk     *dkp;
 618         int             instance;
 619         clock_t         count = 0;

 621         instance = ddi_get_instance(dip);
 622         if (!(dkp = ddi_get_soft_state(cmdk_state, instance)))
 623                 return (DDI_FAILURE);
 624         mutex_enter(&dkp->dk_mutex);
 625         if (dkp->dk_flag & CMDK_SUSPEND) {
 626                 mutex_exit(&dkp->dk_mutex);
 627                 return (DDI_SUCCESS);
 628         }
 629         dkp->dk_flag |= CMDK_SUSPEND;

 631         /* need to wait a while */
 632         while (dadk_getcmds(DKTP_DATA) != 0) {
 633                 delay(drv_sectohz(1));
 633                 delay(drv_usectohz(1000000));
 634                 if (count > 60) {
 635                         dkp->dk_flag &= ~CMDK_SUSPEND;
 636                         cv_broadcast(&dkp->dk_suspend_cv);
 637                         mutex_exit(&dkp->dk_mutex);
 638                         return (DDI_FAILURE);
 639                 }
 640                 count++;
 641         }
 642         mutex_exit(&dkp->dk_mutex);
 643         return (DDI_SUCCESS);
 644 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21086 Wed Aug 19 07:25:25 2015
new/usr/src/uts/intel/io/dktp/hba/ghd/ghd_timer.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 728 /* ************************************************************************ */

 730         /* these are the externally callable routines */


 733 void
 734 ghd_timer_init(tmr_t *tmrp, long ticks)
 735 {
 736         int     indx;

 738         mutex_init(&tglobal_mutex, NULL, MUTEX_DRIVER, NULL);
 739         mutex_init(&tmrp->t_mutex, NULL, MUTEX_DRIVER, NULL);

 741         /*
 742          * determine default timeout value
 743          */
 744         ghd_HZ = drv_sectohz(1);
 744         ghd_HZ = drv_usectohz(1000000);
 745         if (ticks == 0)
 746                 ticks = scsi_watchdog_tick * ghd_HZ;
 747         tmrp->t_ticks = ticks;


 750         /*
 751          * Initialize the table of abort timer values using an
 752          * indirect lookup table so that this code isn't dependant
 753          * on the cmdstate_t enum values or order.
 754          */
 755         for (indx = 0; indx < ghd_ntime_inits; indx++) {
 756                 int     state;
 757                 ulong_t value;

 759                 if (!ghd_time_inits[indx].valid)
 760                         continue;
 761                 state = ghd_time_inits[indx].state;
 762                 value = ghd_time_inits[indx].value;
 763                 ghd_timeout_table[state] = (cmdstate_t)value;
 764         }
 765 }
______unchanged_portion_omitted_
```

```
**********************************************************
   25303 Wed Aug 19 07:25:25 2015
new/usr/src/uts/intel/io/dnet/dnet_mii.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 694 static void
 695 mii_portmon(mii_handle_t mac)
 696 {
 697         int i;
 698         enum mii_phy_state state;
 699         struct phydata *phydata;

 701         /*
 702          * There is a potential deadlock between this test and the
 703          * mutex_enter
 704          */
 705         if (!mac->mii_linknotify) /* Exiting */
 706                 return;

 708         if (mac->lock)
 709                 mutex_enter(mac->lock);

 711         /*
 712          * For each initialised phy, see if the link state has changed, and
 713          * callback to the mac driver if it has
 714          */
 715         for (i = 0; i < 32; i++) {
 716                 if ((phydata = mac->phys[i]) != 0) {
 717                         state = mii_linkup(mac, i) ?
 718                             phy_state_linkup : phy_state_linkdown;
 719                         if (state != phydata->state) {
 720 #ifdef MIIDEBUG
 721                                 if (miidebug)
 722                                         cmn_err(CE_NOTE, "%s: PHY %d link %s",
 723                                             ddi_get_name(mac->mii_dip), i,
 724                                             state == phy_state_linkup ?
 725                                             "up" : "down");
 726 #endif
 727                                 phydata->state = state;
 728                                 mac->mii_linknotify(mac->mii_dip, i, state);
 729                         }
 730                 }
 731         }
 732         /* Check the ports every 5 seconds */
 733         mac->portmon_timer = timeout((void (*)(void*))mii_portmon, (void *)mac,
 734             drv_sectohz(5));
 734             (clock_t)(5 * drv_usectohz(1000000)));
 735         if (mac->lock)
 736                 mutex_exit(mac->lock);
 737 }
______unchanged_portion_omitted_
```

```
**********************************************************
   14876 Wed Aug 19 07:25:25 2015
new/usr/src/uts/intel/io/heci/heci_data_structures.h
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 173 #define LIST_INIT_HEAD  list_init
 174 #define list_del_init(n)        { \
 175             list_del(n); \
 176             list_init(n); \
 177         }

 179 #define list_empty(l)   ((l)->list_next == (l))
 180 #define list_del(p)     { (p)->list_next->list_prev = (p)->list_prev; \
 181         (p)->list_prev->list_next = (p)->list_next; }
 182 #define list_add_tail(newnode, head) { \
 183                 (head)->list_prev->list_next = (newnode); \
 184                 (newnode)->list_prev = (head)->list_prev; \
 185                 (head)->list_prev = (newnode); \
 186                 (newnode)->list_next = (head); \
 187         }
 188 #define list_relink_node(newnode, head) { \
 189                 list_del(newnode); \
 190                 list_add_tail(newnode, head); \
 191         }

 193 #ifdef __GNUC__

 195 #define find_struct(ptr, type, member) ( \
 196         { \
 197         const __typeof(((type *)0)->member) *__tmpp = (ptr);  \
 198         (type *)(void *)((char *)__tmpp - ((size_t)&((type *)0)->member)); \
 199         })
 200 #else
 201 /* type unsafe version */
 202 #define find_struct(ptr, type, member) \
 203                 ((type *)(void *)((char *)(ptr) \
 204                 - ((size_t)&((type *)0)->member)))

 206 #endif

 208 #define list_for_each_entry_safe(pos, n, head, member, type)                \
 209         for (pos = find_struct((head)->list_next, type, member),      \
 210                 n = find_struct(pos->member.list_next, type, member); \
 211                 &pos->member != (head);                                    \
 212                 pos = n, n = find_struct(n->member.list_next, type, member))

 214 #define HZ              drv_sectohz(1)
 214 #define HZ              drv_usectohz(1000000)
 215 /*
 216  * time to wait HECI become ready after init
 217  */
 218 #define HECI_INTEROP_TIMEOUT    (HZ * 7)

 220 /*
 221  * watch dog definition
 222  */
 223 #define HECI_WATCHDOG_DATA_SIZE 16
 224 #define HECI_START_WD_DATA_SIZE 20
 225 #define HECI_WD_PARAMS_SIZE     4
 226 #define HECI_WD_STATE_INDEPENDENCE_MSG_SENT     (1 << 0)

 228 #define HECI_WD_HOST_CLIENT_ID  1
 229 #define HECI_IAMTHIF_HOST_CLIENT_ID     2
```



```
 231 struct guid {
 232         uint32_t data1;
 233         uint16_t data2;
 234         uint16_t data3;
 235         uint8_t data4[8];
 236 };
______unchanged_portion_omitted_
```

****
11556 Wed Aug 19 07:25:26 2015
new/usr/src/uts/intel/io/ipmi/ipmi_kcs.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
175 /*
176  * Start to write a request.  Waits for IBF to clear and then sends the
177  * WR_START command.
178  */
179 static int
180 kcs_start_write(struct ipmi_softc *sc)
181 {
182         int retry, status;

184         for (retry = 0; retry < 10; retry++) {
185                 /* Wait for IBF = 0 */
186                 status = kcs_wait_for_ibf(sc, 0);

188                 /* Clear OBF */
189                 kcs_clear_obf(sc, status);

191                 /* Write start to command */
192                 OUTB(sc, KCS_CTL_STS, KCS_CONTROL_WRITE_START);

194                 /* Wait for IBF = 0 */
195                 status = kcs_wait_for_ibf(sc, 0);
196                 if (KCS_STATUS_STATE(status) == KCS_STATUS_STATE_WRITE)
197                         break;
198                 delay(drv_sectohz(1));
198                 delay(drv_usectohz(1000000));
199         }

201         if (KCS_STATUS_STATE(status) != KCS_STATUS_STATE_WRITE)
202                 /* error state */
203                 return (0);

205         /* Clear OBF */
206         kcs_clear_obf(sc, status);

208         return (1);
209 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  164047 Wed Aug 19 07:25:26 2015
new/usr/src/uts/intel/io/scsi/adapters/arcmsr/arcmsr.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 295 /*
 296  *      Function: arcmsr_attach(9E)
 297  *   Description: Set up all device state and allocate data structures,
 298  *                mutexes, condition variables, etc. for device operation.
 299  *                Set mt_attr property for driver to indicate MT-safety.
 300  *                Add interrupts needed.
 301  *         Input: dev_info_t *dev_info, ddi_attach_cmd_t cmd
 302  *        Output: Return DDI_SUCCESS if device is ready,
 303  *                        else return DDI_FAILURE
 304  */
 305 static int
 306 arcmsr_attach(dev_info_t *dev_info, ddi_attach_cmd_t cmd)
 307 {
 308         scsi_hba_tran_t *hba_trans;
 309         struct ACB *acb;

 311         switch (cmd) {
 312         case DDI_ATTACH:
 313                 return (arcmsr_do_ddi_attach(dev_info,
 314                     ddi_get_instance(dev_info)));
 315         case DDI_RESUME:
 316                 /*
 317                  * There is no hardware state to restart and no
 318                  * timeouts to restart since we didn't DDI_SUSPEND with
 319                  * active cmds or active timeouts We just need to
 320                  * unblock waiting threads and restart I/O the code
 321                  */
 322                 hba_trans = ddi_get_driver_private(dev_info);
 323                 if (hba_trans == NULL) {
 324                         return (DDI_FAILURE);
 325                 }
 326                 acb = hba_trans->tran_hba_private;
 327                 mutex_enter(&acb->acb_mutex);
 328                 arcmsr_iop_init(acb);

 330                 /* restart ccbs "timeout" watchdog */
 331                 acb->timeout_count = 0;
 332                 acb->timeout_id = timeout(arcmsr_ccbs_timeout, (caddr_t)acb,
 333                     drv_sectohz(ARCMSR_TIMEOUT_WATCH));
 333                     (ARCMSR_TIMEOUT_WATCH * drv_usectohz(1000000)));
 334                 acb->timeout_sc_id = timeout(arcmsr_devMap_monitor,
 335                     (caddr_t)acb,
 336                     drv_sectohz(ARCMSR_DEV_MAP_WATCH));
 336                     (ARCMSR_DEV_MAP_WATCH * drv_usectohz(1000000)));
 337                 mutex_exit(&acb->acb_mutex);
 338                 return (DDI_SUCCESS);

 340         default:
 341                 return (DDI_FAILURE);
 342         }
 343 }
______unchanged_portion_omitted_

2779 static int
2780 arcmsr_do_ddi_attach(dev_info_t *dev_info, int instance)
2781 {
2782         scsi_hba_tran_t *hba_trans;
2783         ddi_device_acc_attr_t dev_acc_attr;
```



```
2784         struct ACB *acb;
2785         uint16_t wval;
2786         int raid6 = 1;
2787         char *type;
2788         int intr_types;


2791         /*
2792          * Soft State Structure
2793          * The driver should allocate the per-device-instance
2794          * soft state structure, being careful to clean up properly if
2795          * an error occurs. Allocate data structure.
2796          */
2797         if (ddi_soft_state_zalloc(arcmsr_soft_state, instance) != DDI_SUCCESS) {
2798                 arcmsr_warn(NULL, "ddi_soft_state_zalloc failed");
2799                 return (DDI_FAILURE);
2800         }

2802         acb = ddi_get_soft_state(arcmsr_soft_state, instance);
2803         ASSERT(acb);

2805         arcmsr_mutex_init(acb);

2807         /* acb is already zalloc()d so we don't need to bzero() it */
2808         dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
2809         dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
2810         dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;

2812         acb->dev_info = dev_info;
2813         acb->dev_acc_attr = dev_acc_attr;

2815         /*
2816          * The driver, if providing DMA, should also check that its hardware is
2817          * installed in a DMA-capable slot
2818          */
2819         if (ddi_slaveonly(dev_info) == DDI_SUCCESS) {
2820                 arcmsr_warn(acb, "hardware is not installed in"
2821                     " a DMA-capable slot");
2822                 goto error_level_0;
2823         }
2824         if (pci_config_setup(dev_info, &acb->pci_acc_handle) != DDI_SUCCESS) {
2825                 arcmsr_warn(acb, "pci_config_setup() failed, attach failed");
2826                 goto error_level_0;
2827         }

2829         wval = pci_config_get16(acb->pci_acc_handle, PCI_CONF_VENID);
2830         if (wval != PCI_VENDOR_ID_ARECA) {
2831                 arcmsr_warn(acb,
2832                     "'vendorid (0x%04x) does not match 0x%04x "
2833                     "(PCI_VENDOR_ID_ARECA)",
2834                     wval, PCI_VENDOR_ID_ARECA);
2835                 goto error_level_0;
2836         }

2838         wval = pci_config_get16(acb->pci_acc_handle, PCI_CONF_DEVID);
2839         switch (wval) {
2840         case PCI_DEVICE_ID_ARECA_1110:
2841         case PCI_DEVICE_ID_ARECA_1210:
2842         case PCI_DEVICE_ID_ARECA_1201:
2843                 raid6 = 0;
2844                 /*FALLTHRU*/
2845         case PCI_DEVICE_ID_ARECA_1120:
2846         case PCI_DEVICE_ID_ARECA_1130:
2847         case PCI_DEVICE_ID_ARECA_1160:
2848         case PCI_DEVICE_ID_ARECA_1170:
2849         case PCI_DEVICE_ID_ARECA_1220:
```

```
2850         case PCI_DEVICE_ID_ARECA_1230:
2851         case PCI_DEVICE_ID_ARECA_1260:
2852         case PCI_DEVICE_ID_ARECA_1270:
2853         case PCI_DEVICE_ID_ARECA_1280:
2854                 type = "SATA 3G";
2855                 break;
2856         case PCI_DEVICE_ID_ARECA_1380:
2857         case PCI_DEVICE_ID_ARECA_1381:
2858         case PCI_DEVICE_ID_ARECA_1680:
2859         case PCI_DEVICE_ID_ARECA_1681:
2860                 type = "SAS 3G";
2861                 break;
2862         case PCI_DEVICE_ID_ARECA_1880:
2863                 type = "SAS 6G";
2864                 break;
2865         default:
2866                 type = "X-TYPE";
2867                 arcmsr_warn(acb, "Unknown Host Adapter RAID Controller!");
2868                 goto error_level_0;
2869         }

2871         arcmsr_log(acb, CE_CONT, "Areca %s Host Adapter RAID Controller%s\n",
2872             type, raid6 ? " (RAID6 capable)" : "");

2874         /* we disable iop interrupt here */
2875         if (arcmsr_initialize(acb) == DDI_FAILURE) {
2876                 arcmsr_warn(acb, "arcmsr_initialize failed");
2877                 goto error_level_1;
2878         }

2880         /* Allocate a transport structure */
2881         hba_trans = scsi_hba_tran_alloc(dev_info, SCSI_HBA_CANSLEEP);
2882         if (hba_trans == NULL) {
2883                 arcmsr_warn(acb, "scsi_hba_tran_alloc failed");
2884                 goto error_level_2;
2885         }
2886         acb->scsi_hba_transport = hba_trans;
2887         acb->dev_info = dev_info;
2888         /* init scsi host adapter transport entry */
2889         hba_trans->tran_hba_private  = acb;
2890         hba_trans->tran_tgt_private  = NULL;
2891         /*
2892          * If no per-target initialization is required, the HBA can leave
2893          * tran_tgt_init set to NULL.
2894          */
2895         hba_trans->tran_tgt_init = arcmsr_tran_tgt_init;
2896         hba_trans->tran_tgt_probe = scsi_hba_probe;
2897         hba_trans->tran_tgt_free = NULL;
2898         hba_trans->tran_start = arcmsr_tran_start;
2899         hba_trans->tran_abort = arcmsr_tran_abort;
2900         hba_trans->tran_reset = arcmsr_tran_reset;
2901         hba_trans->tran_getcap = arcmsr_tran_getcap;
2902         hba_trans->tran_setcap = arcmsr_tran_setcap;
2903         hba_trans->tran_init_pkt = arcmsr_tran_init_pkt;
2904         hba_trans->tran_destroy_pkt = arcmsr_tran_destroy_pkt;
2905         hba_trans->tran_dmafree = arcmsr_tran_dmafree;
2906         hba_trans->tran_sync_pkt = arcmsr_tran_sync_pkt;

2908         hba_trans->tran_reset_notify = NULL;
2909         hba_trans->tran_get_bus_addr = NULL;
2910         hba_trans->tran_get_name = NULL;
2911         hba_trans->tran_quiesce = NULL;
2912         hba_trans->tran_unquiesce = NULL;
2913         hba_trans->tran_bus_reset = NULL;
2914         hba_trans->tran_bus_config = arcmsr_tran_bus_config;
2915         hba_trans->tran_add_eventcall = NULL;
```



```
2916         hba_trans->tran_get_eventcookie = NULL;
2917         hba_trans->tran_post_event = NULL;
2918         hba_trans->tran_remove_eventcall = NULL;

2920         /* iop init and enable interrupt here */
2921         arcmsr_iop_init(acb);

2923         /* Get supported interrupt types */
2924         if (ddi_intr_get_supported_types(dev_info, &intr_types) !=
2925             DDI_SUCCESS) {
2926                 arcmsr_warn(acb, "ddi_intr_get_supported_types failed");
2927                 goto error_level_3;
2928         }
2929         if (intr_types & DDI_INTR_TYPE_FIXED) {
2930                 if (arcmsr_add_intr(acb, DDI_INTR_TYPE_FIXED) != DDI_SUCCESS)
2931                         goto error_level_5;
2932         } else if (intr_types & DDI_INTR_TYPE_MSI) {
2933                 if (arcmsr_add_intr(acb, DDI_INTR_TYPE_FIXED) != DDI_SUCCESS)
2934                         goto error_level_5;
2935         }

2937         /*
2938          * The driver should attach this instance of the device, and
2939          * perform error cleanup if necessary
2940          */
2941         if (scsi_hba_attach_setup(dev_info, &arcmsr_dma_attr,
2942             hba_trans, SCSI_HBA_TRAN_CLONE) != DDI_SUCCESS) {
2943                 arcmsr_warn(acb, "scsi_hba_attach_setup failed");
2944                 goto error_level_5;
2945         }

2947         /* Create a taskq for dealing with dr events */
2948         if ((acb->taskq = ddi_taskq_create(dev_info, "arcmsr_dr_taskq", 1,
2949             TASKQ_DEFAULTPRI, 0)) == NULL) {
2950                 arcmsr_warn(acb, "ddi_taskq_create failed");
2951                 goto error_level_8;
2952         }

2954         acb->timeout_count = 0;
2955         /* active ccbs "timeout" watchdog */
2956         acb->timeout_id = timeout(arcmsr_ccbs_timeout, (caddr_t)acb,
2957             drv_sectohz(ARCMSR_TIMEOUT_WATCH));
2957             (ARCMSR_TIMEOUT_WATCH * drv_usectohz(1000000)));
2958         acb->timeout_sc_id = timeout(arcmsr_devMap_monitor, (caddr_t)acb,
2959             drv_sectohz(ARCMSR_DEV_MAP_WATCH));
2959             (ARCMSR_DEV_MAP_WATCH * drv_usectohz(1000000)));

2961         /* report device info */
2962         ddi_report_dev(dev_info);

2964         return (DDI_SUCCESS);

2966 error_level_8:

2968 error_level_7:
2969 error_level_6:
2970         (void) scsi_hba_detach(dev_info);

2972 error_level_5:
2973         arcmsr_remove_intr(acb);

2975 error_level_3:
2976 error_level_4:
2977         if (acb->scsi_hba_transport)
2978                 scsi_hba_tran_free(acb->scsi_hba_transport);
```

```
2980 error_level_2:
2981         if (acb->ccbs_acc_handle)
2982                 ddi_dma_mem_free(&acb->ccbs_acc_handle);
2983         if (acb->ccbs_pool_handle)
2984                 ddi_dma_free_handle(&acb->ccbs_pool_handle);

2986 error_level_1:
2987         if (acb->pci_acc_handle)
2988                 pci_config_teardown(&acb->pci_acc_handle);
2989         arcmsr_mutex_destroy(acb);
2990         ddi_soft_state_free(arcmsr_soft_state, instance);

2992 error_level_0:
2993         return (DDI_FAILURE);
2994 }
```

*unchanged_portion_omitted_*

```
3145 static void
3146 arcmsr_ccbs_timeout(void* arg)
3147 {
3148         struct ACB *acb = (struct ACB *)arg;
3149         struct CCB *ccb;
3150         int i, instance, timeout_count = 0;
3151         uint32_t intmask_org;
3152         time_t current_time = ddi_get_time();

3154         intmask_org = arcmsr_disable_allintr(acb);
3155         mutex_enter(&acb->isr_mutex);
3156         if (acb->ccboutstandingcount != 0) {
3157                 /* check each ccb */
3158                 i = ddi_dma_sync(acb->ccbs_pool_handle, 0, 0,
3159                     DDI_DMA_SYNC_FORKERNEL);
3160                 if (i != DDI_SUCCESS) {
3161                         if ((acb->timeout_id != 0) &&
3162                             ((acb->acb_flags & ACB_F_SCSISTOPADAPTER) == 0)) {
3163                                 /* do pkt timeout check each 60 secs */
3164                                 acb->timeout_id = timeout(arcmsr_ccbs_timeout,
3165                                     (void*)acb, drv_sectohz(ARCMSR_TIMEOUT_WATCH
3165                                     (void*)acb, (ARCMSR_TIMEOUT_WATCH *
3166                                     drv_usectohz(1000000)));
3166                         }
3167                         mutex_exit(&acb->isr_mutex);
3168                         arcmsr_enable_allintr(acb, intmask_org);
3169                         return;
3170                 }
3171                 instance = ddi_get_instance(acb->dev_info);
3172                 for (i = 0; i < ARCMSR_MAX_FREECCB_NUM; i++) {
3173                         ccb = acb->pccb_pool[i];
3174                         if (ccb->acb != acb) {
3175                                 break;
3176                         }
3177                         if (ccb->ccb_state == ARCMSR_CCB_FREE) {
3178                                 continue;
3179                         }
3180                         if (ccb->pkt == NULL) {
3181                                 continue;
3182                         }
3183                         if (ccb->pkt->pkt_time == 0) {
3184                                 continue;
3185                         }
3186                         if (ccb->ccb_time >= current_time) {
3187                                 continue;
3188                         }
3189                         int id = ccb->pkt->pkt_address.a_target;
3190                         int lun = ccb->pkt->pkt_address.a_lun;
```



```
3191                         if (ccb->ccb_state == ARCMSR_CCB_START) {
3192                                 uint8_t *cdb = (uint8_t *)&ccb->arcmsr_cdb.Cdb;

3194                                 timeout_count++;
3195                                 arcmsr_warn(acb,
3196                                     "scsi target %d lun %d cmd=0x%x "
3197                                     "command timeout, ccb=0x%p",
3198                                     instance, id, lun, *cdb, (void *)ccb);
3199                                 ccb->ccb_state = ARCMSR_CCB_TIMEOUT;
3200                                 ccb->pkt->pkt_reason = CMD_TIMEOUT;
3201                                 ccb->pkt->pkt_statistics = STAT_TIMEOUT;
3202                                 /* acb->devstate[id][lun] = ARECA_RAID_GONE; */
3203                                 arcmsr_ccb_complete(ccb, 1);
3204                                 continue;
3205                         } else if ((ccb->ccb_state & ARCMSR_CCB_CAN_BE_FREE) ==
3206                             ARCMSR_CCB_CAN_BE_FREE) {
3207                                 arcmsr_free_ccb(ccb);
3208                         }
3209                 }
3210         }
3211         if ((acb->timeout_id != 0) &&
3212             ((acb->acb_flags & ACB_F_SCSISTOPADAPTER) == 0)) {
3213                 /* do pkt timeout check each 60 secs */
3214                 acb->timeout_id = timeout(arcmsr_ccbs_timeout,
3215                     (void*)acb, drv_sectohz(ARCMSR_TIMEOUT_WATCH));
3216                     (void*)acb, (ARCMSR_TIMEOUT_WATCH * drv_usectohz(1000000)));
3216         }
3217         mutex_exit(&acb->isr_mutex);
3218         arcmsr_enable_allintr(acb, intmask_org);
3219 }
```

*unchanged_portion_omitted_*

```
3385 static void
3386 arcmsr_devMap_monitor(void* arg)
3387 {

3389         struct ACB *acb = (struct ACB *)arg;
3390         switch (acb->adapter_type) {
3391         case ACB_ADAPTER_TYPE_A:
3392         {
3393                 struct HBA_msgUnit *phbamu;

3395                 phbamu = (struct HBA_msgUnit *)acb->pmu;
3396                 CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
3397                     &phbamu->inbound_msgaddr0,
3398                     ARCMSR_INBOUND_MESG0_GET_CONFIG);
3399                 break;
3400         }

3402         case ACB_ADAPTER_TYPE_B:
3403         {
3404                 struct HBB_msgUnit *phbbmu;

3406                 phbbmu = (struct HBB_msgUnit *)acb->pmu;
3407                 CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
3408                     &phbbmu->hbb_doorbell->drv2iop_doorbell,
3409                     ARCMSR_MESSAGE_GET_CONFIG);
3410                 break;
3411         }

3413         case ACB_ADAPTER_TYPE_C:
3414         {
3415                 struct HBC_msgUnit *phbcmu;

3417                 phbcmu = (struct HBC_msgUnit *)acb->pmu;
```

```
3418                 CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
3419                     &phbcmu->inbound_msgaddr0,
3420                     ARCMSR_INBOUND_MESG0_GET_CONFIG);
3421                 CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
3422                     &phbcmu->inbound_doorbell,
3423                     ARCMSR_HBCMU_DRV2IOP_MESSAGE_CMD_DONE);
3424                 break;
3425         }

3427         }

3429         if ((acb->timeout_id != 0) &&
3430             ((acb->acb_flags & ACB_F_SCSISTOPADAPTER) == 0)) {
3431                 /* do pkt timeout check each 5 secs */
3432                 acb->timeout_id = timeout(arcmsr_devMap_monitor, (void*)acb,
3433                     drv_sectohz(ARCMSR_DEV_MAP_WATCH));
3434                     (ARCMSR_DEV_MAP_WATCH * drv_usectohz(1000000)));
3434         }
3435 }
```

_____***unchanged_portion_omitted_***

****
14948 Wed Aug 19 07:25:26 2015
new/usr/src/uts/sparc/io/pciex/pcieb_sparc.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 474 #ifdef  PRINT_PLX_SEEPROM_CRC
 475 static void
 476 pcieb_print_plx_seeprom_crc_data(pcieb_devstate_t *pcieb_p)
 477 {
 478         ddi_acc_handle_t h;
 479         dev_info_t *dip = pcieb_p->pcieb_dip;
 480         uint16_t vendorid = (PCIE_DIP2BUS(dip)->bus_dev_ven_id) & 0xFFFF;
 481         int nregs;
 482         caddr_t mp;
 483         off_t bar_size;
 484         ddi_device_acc_attr_t mattr = {
 485                 DDI_DEVICE_ATTR_V0,
 486                 DDI_STRUCTURE_LE_ACC,
 487                 DDI_STRICTORDER_ACC
 488         };
 489         uint32_t addr_reg_off = 0x260, data_reg_off = 0x264, data = 0x6BE4;

 491         if (vendorid != PXB_VENDOR_PLX)
 492                 return;
 493         if (ddi_dev_nregs(dip, &nregs) != DDI_SUCCESS)
 494                 return;
 495         if (nregs < 2)  /* check for CONF entry only, no BARs */
 496                 return;
 497         if (ddi_dev_regsize(dip, 1, &bar_size) != DDI_SUCCESS)
 498                 return;
 499         if (ddi_regs_map_setup(dip, 1, (caddr_t *)&mp, 0, bar_size,
 500             &mattr, &h) != DDI_SUCCESS)
 501                 return;
 502         ddi_put32(h, (uint32_t *)((uchar_t *)mp + addr_reg_off), data);
 503         delay(drv_sectohz(1));
 503         delay(drv_usectohz(1000000));
 504         printf("%s#%d: EEPROM StatusReg = %x, CRC = %x\n",
 505             ddi_driver_name(dip), ddi_get_instance(dip),
 506             ddi_get32(h, (uint32_t *)((uchar_t *)mp + addr_reg_off)),
 507             ddi_get32(h, (uint32_t *)((uchar_t *)mp + data_reg_off)));
 508 #ifdef PLX_HOT_RESET_DISABLE
 509         /* prevent hot reset from propogating downstream. */
 510         data = ddi_get32(h, (uint32_t *)((uchar_t *)mp + 0x1DC));
 511         ddi_put32(h, (uint32_t *)((uchar_t *)mp + 0x1DC), data | 0x80000);
 512         delay(drv_sectohz(1));
 512         delay(drv_usectohz(1000000));
 513         printf("%s#%d: EEPROM 0x1DC prewrite=%x postwrite=%x\n",
 514             ddi_driver_name(dip), ddi_get_instance(dip), data,
 515             ddi_get32(h, (uint32_t *)((uchar_t *)mp + 0x1DC)));
 516 #endif /* PLX_HOT_RESET_DISABLE */
 517         ddi_regs_map_free(&h);
 518 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  115015 Wed Aug 19 07:25:26 2015
new/usr/src/uts/sun/io/dada/targets/dad.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 667 static int
 668 dcddetach(dev_info_t *devi, ddi_detach_cmd_t cmd)
 669 {
 670         int instance;
 671         struct dcd_disk *un;
 672         clock_t wait_cmds_complete;
 673         instance = ddi_get_instance(devi);

 675         if (!(un = ddi_get_soft_state(dcd_state, instance)))
 676                 return (DDI_FAILURE);

 678         switch (cmd) {
 679         case DDI_DETACH:
 680                 return (dcd_dr_detach(devi));

 682         case DDI_SUSPEND:
 683                 mutex_enter(DCD_MUTEX);
 684                 if (un->un_state == DCD_STATE_SUSPENDED) {
 685                         mutex_exit(DCD_MUTEX);
 686                         return (DDI_SUCCESS);
 687                 }
 688                 un->un_throttle = 0;
 689                 /*
 690                  * Save the last state first
 691                  */
 692                 un->un_save_state = un->un_last_state;

 694                 New_state(un, DCD_STATE_SUSPENDED);

 696                 /*
 697                  * wait till current operation completed. If we are
 698                  * in the resource wait state (with an intr outstanding)
 699                  * then we need to wait till the intr completes and
 700                  * starts the next cmd. We wait for
 701                  * DCD_WAIT_CMDS_COMPLETE seconds before failing the
 702                  * DDI_SUSPEND.
 703                  */
 704                 wait_cmds_complete = ddi_get_lbolt();
 705                 wait_cmds_complete +=
 706                     drv_sectohz(DCD_WAIT_CMDS_COMPLETE);
 706                     DCD_WAIT_CMDS_COMPLETE * drv_usectohz(1000000);

 708                 while (un->un_ncmds) {
 709                         if (cv_timedwait(&un->un_disk_busy_cv,
 710                             DCD_MUTEX, wait_cmds_complete) == -1) {
 711                                 /*
 712                                  * commands Didn't finish in the
 713                                  * specified time, fail the DDI_SUSPEND.
 714                                  */
 715                                 DAD_DEBUG2(DCD_DEVINFO, dcd_label,
 716                                     DCD_DEBUG, "dcddetach: SUSPEND "
 717                                     "failed due to outstanding cmds\n");
 718                                 Restore_state(un);
 719                                 mutex_exit(DCD_MUTEX);
 720                                 return (DDI_FAILURE);
 721                         }
 722                 }
 723                 mutex_exit(DCD_MUTEX);
 724                 return (DDI_SUCCESS);
```



```
 725         }
 726         return (DDI_FAILURE);
 727 }
______unchanged_portion_omitted_
```

```
**********************************************************
  164833 Wed Aug 19 07:25:27 2015
new/usr/src/uts/sun/io/fd.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3577 /*
3578  * fdexec
3579  *      all commands go through here.  Assumes the command block
3580  *      fdctlr.c_csb is filled in.  The bytes are sent to the
3581  *      controller and then we do whatever else the csb says -
3582  *      like wait for immediate results, etc.
3583  *
3584  *      All waiting for operations done is in here - to allow retrys
3585  *      and checking for disk changed - so we don't have to worry
3586  *      about sleeping at interrupt level.
3587  *
3588  * RETURNS: 0 if all ok,
3589  *      ENXIO - diskette not in drive
3590  *      EBUSY - if chip is locked or busy
3591  *      EIO - for timeout during sending cmds to chip
3592  *
3593  * to sleep: set FDXC_SLEEP, to check for disk
3594  * changed: set FDXC_CHECKCHG
3595  *
3596  *      - called with the lock held
3597  */
3598 static int
3599 fdexec(struct fdctlr *fdc, int flags)
3600 {
3601         struct fdcsb *csb;
3602         int     i;
3603         int     to, unit;
3604         uchar_t tmp;
3605         caddr_t a = (caddr_t)fdc;

3607         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: flags:%x\n", flags));

3609         ASSERT(mutex_owned(&fdc->c_lolock));

3611         csb = &fdc->c_csb;
3612         unit = csb->csb_unit;


3615         ASSERT(unit == fdc->c_un->un_unit_no);

3617 retry:
3618         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: cmd is %s\n",
3619             fdcmds[csb->csb_cmds[0] & 0x1f].cmdname));
3620         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: transfer rate = %d\n",
3621             fdc->c_un->un_chars->fdc_transfer_rate));
3622         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: sec size = %d\n",
3623             fdc->c_un->un_chars->fdc_sec_size));
3624         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: nblocks (512) = %d\n",
3625             fdc->c_un->un_label.dkl_map[2].dkl_nblk));

3627         if ((fdc->c_fdtype & FDCTYPE_CTRLMASK) == FDCTYPE_82077) {
3628                 fdexec_turn_on_motor(fdc, flags, unit);
3629         }


3632         fdselect(fdc, unit, 1); /* select drive */

3634         /*
3635          * select data rate for this unit/command
```



```
3636          */
3637         switch (fdc->c_un->un_chars->fdc_transfer_rate) {
3638         case 500:
3639                 Dsr(fdc, 0);
3640                 break;
3641         case 300:
3642                 Dsr(fdc, 1);
3643                 break;
3644         case 250:
3645                 Dsr(fdc, 2);
3646                 break;
3647         }
3648         drv_usecwait(2);


3651         /*
3652          * If checking for changed is enabled (i.e., not seeking in checkdisk),
3653          * we sample the DSKCHG line to see if the diskette has wandered away.
3654          */
3655         if ((flags & FDXC_CHECKCHG) && fdsense_chng(fdc, unit)) {
3656                 FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "diskette changed\n"));
3657                 fdc->c_un->un_flags |= FDUNIT_CHANGED;

3659                 if (fdcheckdisk(fdc, unit)) {

3661                         (void) fd_unbind_handle(fdc);
3662                         return (ENXIO);

3664                 }
3665         }

3667         /*
3668          * gather some statistics
3669          */
3670         switch (csb->csb_cmds[0] & 0x1f) {
3671         case FDRAW_RDCMD:
3672                 fdc->fdstats.rd++;
3673                 break;
3674         case FDRAW_WRCMD:
3675                 fdc->fdstats.wr++;
3676                 break;
3677         case FDRAW_REZERO:
3678                 fdc->fdstats.recal++;
3679                 break;
3680         case FDRAW_FORMAT:
3681                 fdc->fdstats.form++;
3682                 break;
3683         default:
3684                 fdc->fdstats.other++;
3685                 break;
3686         }

3688         /*
3689          * Always set the opmode *prior* to poking the chip.
3690          * This way we don't have to do any locking at high level.
3691          */
3692         csb->csb_raddr = 0;
3693         csb->csb_rlen = 0;
3694         if (csb->csb_opflags & CSB_OFSEEKOPS) {
3695                 csb->csb_opmode = 2;
3696         } else if (csb->csb_opflags & CSB_OFIMMEDIATE) {
3697                 csb->csb_opmode = 0;
3698         } else {
3699                 csb->csb_opmode = 1;    /* normal data xfer commands */
3700                 csb->csb_raddr = csb->csb_addr;
3701                 csb->csb_rlen = csb->csb_len;
```

```
3702        }

3704        bzero((caddr_t)csb->csb_rslt, 10);
3705        csb->csb_status = 0;
3706        csb->csb_cmdstat = 0;


3709        /*
3710         * Program the DMA engine with the length and address of the transfer
3711         * (DMA is only used on a read or a write)
3712         */
3713        if ((fdc->c_fdtype & FDCTYPE_DMA) &&
3714            ((fdc->c_csb.csb_read == CSB_READ) ||
3715            (fdc->c_csb.csb_read == CSB_WRITE)))  {
3716                mutex_enter(&fdc->c_hilock);

3718                /* Reset the dcsr to clear it of all errors */

3720                reset_dma_controller(fdc);

3722                FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "cookie addr 0x%p\n",
3723                    (void *)fdc->c_csb.csb_dmacookie.dmac_laddress));

3725                FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "cookie length %ld\n",
3726                    fdc->c_csb.csb_dmacookie.dmac_size));
3727                ASSERT(fdc->c_csb.csb_dmacookie.dmac_size);

3729                set_data_count_register(fdc,
3730                    fdc->c_csb.csb_dmacookie.dmac_size);
3731                set_data_address_register(fdc,
3732                    fdc->c_csb.csb_dmacookie.dmac_laddress);

3734                /* Program the DCSR */

3736                if (fdc->c_csb.csb_read == CSB_READ)
3737                        set_dma_mode(fdc, CSB_READ);
3738                else
3739                        set_dma_mode(fdc, CSB_WRITE);
3740                mutex_exit(&fdc->c_hilock);
3741        }

3743        /*
3744         * I saw this (chip unexpectedly busy) happen when i shoved the
3745         * floppy into the drive while
3746         * running a dd if= /dev/rfd0c. so it *is* possible for this to happen.
3747         * we need to do a ctlr reset ...
3748         */

3750        if (Msr(fdc) & CB) {
3751                /* tried to give command to chip when it is busy! */
3752                FDERRPRINT(FDEP_L3, FDEM_EXEC,
3753                    (C, "fdc: unexpectedly busy-stat 0x%x\n", Msr(fdc)));
3754                csb->csb_cmdstat = 1;   /* XXX TBD ERRS NYD for now */

3756                (void) fd_unbind_handle(fdc);
3757                return (EBUSY);
3758        }

3760        /* Give command to the controller */
3761        for (i = 0; i < (int)csb->csb_ncmds; i++) {

3763                /* Test the readiness of the controller to receive the cmd */
3764                for (to = FD_CRETRY; to; to--) {
3765                        if ((Msr(fdc) & (DIO|RQM)) == RQM)
3766                                break;
3767                }
```



```
3768                if (to == 0) {
3769                        FDERRPRINT(FDEP_L2, FDEM_EXEC,
3770                            (C, "fdc: no RQM - stat 0x%x\n", Msr(fdc)));
3771                        csb->csb_cmdstat = 1;

3773                        (void) fd_unbind_handle(fdc);
3774                        return (EIO);
3775                }

3777                Set_Fifo(fdc, csb->csb_cmds[i]);

3779                FDERRPRINT(FDEP_L1, FDEM_EXEC,
3780                    (C, "fdexec: sent 0x%x, Msr 0x%x\n", csb->csb_cmds[i],
3781                    Msr(fdc)));

3783        }


3786        /*
3787         * Start watchdog timer on data transfer type commands - required
3788         * in case a diskette is not present or is unformatted
3789         */
3790        if (csb->csb_opflags & CSB_OFTIMEIT) {
3791                fdc->c_timeid = timeout(fdwatch, a,
3792                    drv_sectohz(tosec));
3792                    tosec * drv_usectohz(1000000));
3793        }

3795        FDERRPRINT(FDEP_L1, FDEM_EXEC,
3796            (C, "fdexec: cmd sent, Msr 0x%x\n", Msr(fdc)));

3798        /* If the operation has no results - then just return */
3799        if (csb->csb_opflags & CSB_OFNORESULTS) {
3800                if (fdc->c_fdtype & FDCTYPE_82077) {
3801                        if (fdc->c_mtimeid == 0) {
3802                                fdc->c_mtimeid = timeout(fdmotoff, a,
3803                                    Motoff_delay);
3804                        }
3805                }
3806                FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: O K ..\n"));

3808                /*
3809                 * Make sure the last byte is received well by the
3810                 * controller. On faster CPU, it may still be busy
3811                 * by the time another command comes here.
3812                 */
3813                for (to = FD_CRETRY; to; to--) {
3814                        if ((Msr(fdc) & (DIO|RQM)) == RQM)
3815                                break;
3816                        }
3817                if (to == 0) {
3818                        csb->csb_cmdstat = 1;
3819                        return (EIO);
3820                }

3822                /*
3823                 * An operation that has no results isn't doing DMA so,
3824                 * there is no reason to try to unbind a handle
3825                 */
3826                return (0);
3827        }

3829        /*
3830         * If this operation has no interrupt AND an immediate result
3831         * then we just busy wait for the results and stuff them into
3832         * the csb
```

```
3833          */
3834         if (csb->csb_opflags & CSB_OFIMMEDIATE) {
3835                 to = FD_RRETRY;
3836                 csb->csb_nrslts = 0;
3837                 /*
3838                  * Wait while this command is still going on.
3839                  */
3840                 while ((tmp = Msr(fdc)) & CB) {
3841                         /*
3842                          * If RQM + DIO, then a result byte is at hand.
3843                          */
3844                         if ((tmp & (RQM|DIO|CB)) == (RQM|DIO|CB)) {
3845                                 csb->csb_rslt[csb->csb_nrslts++] =
3846                                     Fifo(fdc);
3847                                 /*
3848                                  * FDERRPRINT(FDEP_L4, FDEM_EXEC,
3849                                  *    (C, "fdexec: got result 0x%x\n",
3850                                  *    csb->csb_nrslts));
3851                                  */
3852                         } else if (--to == 0) {
3853                                 FDERRPRINT(FDEP_L4, FDEM_EXEC,
3854                                     (C, "fdexec: timeout, Msr%x, nr%x\n",
3855                                     Msr(fdc), csb->csb_nrslts));

3857                                 csb->csb_status = 2;
3858                                 if (fdc->c_fdtype & FDCTYPE_82077) {
3859                                         if (fdc->c_mtimeid == 0) {
3860                                                 fdc->c_mtimeid = timeout(
3861                                                     fdmotoff, a, Motoff_delay);
3862                                         }
3863                                 }
3864                                 /*
3865                                  * There is no DMA happening.  No need to
3866                                  * try freeing a handle.
3867                                  */

3869                                 return (EIO);
3870                         }
3871                 }
3872         }

3874         /*
3875          * If told to sleep here, well then sleep!
3876          */

3878         if (flags & FDXC_SLEEP) {
3879                 fdc->c_flags |= FDCFLG_WAITING;
3880                 while (fdc->c_flags & FDCFLG_WAITING) {
3881                         cv_wait(&fdc->c_iocv, &fdc->c_lolock);
3882                 }
3883         }

3885         /*
3886          * kludge for end-of-cylinder error which must be ignored!!!
3887          */

3889         if ((fdc->c_fdtype & FDCTYPE_TCBUG) &&
3890             ((csb->csb_rslt[0] & IC_SR0) == 0x40) &&
3891             (csb->csb_rslt[1] & EN_SR1))
3892                 csb->csb_rslt[0] &= ~IC_SR0;

3894         /*
3895          * See if there was an error detected, if so, fdrecover()
3896          * will check it out and say what to do.
3897          *
3898          * Don't do this, though, if this was the Sense Drive Status
```



```
3899          * or the Dump Registers command.
3900          */
3901         if (((csb->csb_rslt[0] & IC_SR0) || (fdc->c_csb.csb_dcsr_rslt) ||
3902             (csb->csb_status)) &&
3903             ((csb->csb_cmds[0] != FDRAW_SENSE_DRV) &&
3904             (csb->csb_cmds[0] != DUMPREG))) {
3905                 /* if it can restarted OK, then do so, else return error */
3906                 if (fdrecover(fdc) != 0) {
3907                         if (fdc->c_fdtype & FDCTYPE_82077) {
3908                                 if (fdc->c_mtimeid == 0) {
3909                                         fdc->c_mtimeid = timeout(fdmotoff,
3910                                             a, Motoff_delay);
3911                                 }
3912                         }

3914                         /*
3915                          * If this was a dma transfer, unbind the handle so
3916                          * that other transfers may use it.
3917                          */

3919                         (void) fd_unbind_handle(fdc);
3920                         return (EIO);
3921                 } else {
3922                         /* ASSUMES that cmd is still intact in csb */
3923                         goto retry;
3924                 }
3925         }

3927         /* things went ok */
3928         if (fdc->c_fdtype & FDCTYPE_82077) {
3929                 if (fdc->c_mtimeid == 0) {
3930                         fdc->c_mtimeid = timeout(fdmotoff, a, Motoff_delay);
3931                 }
3932         }
3933         FDERRPRINT(FDEP_L1, FDEM_EXEC, (C, "fdexec: O K ..........\n"));

3935         if (fd_unbind_handle(fdc))
3936                 return (EIO);

3938         return (0);
3939 }
```

*unchanged_portion_omitted_*

```
4756 /*
4757  * fdreset
4758  *      reset THE controller, and configure it to be
4759  *      the way it ought to be
4760  * ASSUMES: that it already owns the csb/fdctlr!
4761  *
4762  *      - called with the low level lock held
4763  */
4764 static int
4765 fdreset(struct fdctlr *fdc)
4766 {
4767         struct fdcsb *csb;
4768         clock_t local_lbolt = 0;
4769         timeout_id_t timeid;

4771         FDERRPRINT(FDEP_L1, FDEM_RESE, (C, "fdreset\n"));

4773         ASSERT(mutex_owned(&fdc->c_lolock));

4775         /* count resets */
4776         fdc->fdstats.reset++;
```

```
4778         /*
4779          * On the 82077, the DSR will clear itself after a reset.  Upon exiting
4780          * the reset, a polling interrupt will be generated.  If the floppy
4781          * interrupt is enabled, it's possible for cv_signal() to be called
4782          * before cv_wait().  This will cause the system to hang.  Turn off
4783          * the floppy interrupt to avoid this race condition
4784          */
4785         if ((fdc->c_fdtype & FDCTYPE_CTRLMASK) == FDCTYPE_82077) {
4786                 /*
4787                  * We need to perform any timeouts before we Reset the
4788                  * controller. We cannot afford to drop the c_lolock mutex after
4789                  * Resetting the controller. The reason is that we get a spate
4790                  * of interrupts until we take the controller out of reset.
4791                  * The way we avoid this spate of continuous interrupts is by
4792                  * holding on to the c_lolock and forcing the fdintr_dma routine
4793                  * to go to sleep waiting for this mutex.
4794                  */
4795                 /* Do not hold the mutex across the untimeout call */
4796                 timeid = fdc->c_mtimeid;
4797                 fdc->c_mtimeid = 0;
4798                 if (timeid) {
4799                         mutex_exit(&fdc->c_lolock);
4800                         (void) untimeout(timeid);
4801                         mutex_enter(&fdc->c_lolock);
4802                 }
4803                 /* LINTED */
4804                 Set_dor(fdc, DMAGATE, 0);
4805                 FDERRPRINT(FDEP_L1, FDEM_RESE, (C, "fdreset: set dor\n"));
4806         }

4808         /* toggle software reset */
4809         Dsr(fdc, SWR);

4811         drv_usecwait(5);

4813         FDERRPRINT(FDEP_L1, FDEM_RESE,
4814             (C, "fdreset: toggled software reset\n"));

4816         /*
4817          * This sets the data rate to 500Kbps (for high density)
4818          * XXX should use current characteristics instead XXX
4819          */
4820         Dsr(fdc, 0);
4821         drv_usecwait(5);
4822         switch (fdc->c_fdtype & FDCTYPE_CTRLMASK) {
4823         case FDCTYPE_82077:
4824                 /*
4825                  * when we bring the controller out of reset it will generate
4826                  * a polling interrupt. fdintr() will field it and schedule
4827                  * fd_lointr(). There will be no one sleeping but we are
4828                  * expecting an interrupt so....
4829                  */
4830                 fdc->c_flags |= FDCFLG_WAITING;

4832                 /*
4833                  * The reset bit must be cleared to take the 077 out of
4834                  * reset state and the DMAGATE bit must be high to enable
4835                  * interrupts.
4836                  */
4837                 /* LINTED */
4838                 Set_dor(fdc, DMAGATE|RESET, 1);

4840                 FDERRPRINT(FDEP_L1, FDEM_ATTA,
4841                     (C, "fdattach: Dor 0x%x\n", Dor(fdc)));

4843                 local_lbolt = ddi_get_lbolt();
```



```
4844                 if (cv_timedwait(&fdc->c_iocv, &fdc->c_lolock,
4845                     local_lbolt + drv_sectohz(1)) == -1) {
4845                     local_lbolt + drv_usectohz(1000000)) == -1) {
4846                         return (-1);
4847                 }
4848                 break;

4850         default:
4851                 fdc->c_flags |= FDCFLG_WAITING;

4853                 /*
4854                  * A timed wait is not used because it's possible for the timer
4855                  * to go off before the controller has a chance to interrupt.
4856                  */
4857                 cv_wait(&fdc->c_iocv, &fdc->c_lolock);
4858                 break;
4859         }
4860         csb = &fdc->c_csb;

4862         /* setup common things in csb */
4863         csb->csb_unit = fdc->c_un->un_unit_no;
4864         csb->csb_nrslts = 0;
4865         csb->csb_opflags = CSB_OFNORESULTS;
4866         csb->csb_maxretry = 0;
4867         csb->csb_retrys = 0;

4869         csb->csb_read = CSB_NULL;

4871         /* send SPECIFY command to fdc */
4872         /* csb->unit is don't care */
4873         csb->csb_cmds[0] = FDRAW_SPECIFY;
4874         csb->csb_cmds[1] = fdspec[0]; /* step rate, head unload time */
4875         if (fdc->c_fdtype & FDCTYPE_DMA)
4876                 csb->csb_cmds[2] =  SPEC_DMA_MODE;
4877         else
4878                 csb->csb_cmds[2] = fdspec[1];  /* head load time, DMA mode */

4880         csb->csb_ncmds = 3;

4882         /* XXX for now ignore errors, they "CAN'T HAPPEN" */
4883         (void) fdexec(fdc, 0);  /* no FDXC_CHECKCHG, ... */
4884         /* no results */

4886         /* send CONFIGURE command to fdc */
4887         /* csb->unit is don't care */
4888         csb->csb_cmds[0] = CONFIGURE;
4889         csb->csb_cmds[1] = fdconf[0]; /* motor info, motor delays */
4890         csb->csb_cmds[2] = fdconf[1]; /* enaimplsk, disapoll, fifothru */
4891         csb->csb_cmds[3] = fdconf[2]; /* track precomp */
4892         csb->csb_ncmds = 4;

4894         csb->csb_read = CSB_NULL;

4896         csb->csb_retrys = 0;

4898         /* XXX for now ignore errors, they "CAN'T HAPPEN" */
4899         (void) fdexec(fdc, 0); /* no FDXC_CHECKCHG, ... */
4900         return (0);
4901 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  245838 Wed Aug 19 07:25:27 2015
new/usr/src/uts/sun/io/scsi/adapters/fas.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 520 /*ARGSUSED*/
 521 static int
 522 fas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 523 {
 524         struct fas      *fas = NULL;
 525         volatile struct dma     *dmar = NULL;
 526         volatile struct fasreg  *fasreg;
 527         ddi_dma_attr_t          *fas_dma_attr;
 528         ddi_device_acc_attr_t   dev_attr;

 530         int                     instance, id, slot, i, hm_rev;
 531         size_t                  rlen;
 532         uint_t                  count;
 533         char                    buf[64];
 534         scsi_hba_tran_t         *tran = NULL;
 535         char                    intr_added = 0;
 536         char                    mutex_init_done = 0;
 537         char                    hba_attached = 0;
 538         char                    bound_handle = 0;
 539         char                    *prop_template = "target%d-scsi-options";
 540         char                    prop_str[32];

 542         /* CONSTCOND */
 543         ASSERT(NO_COMPETING_THREADS);

 545         switch (cmd) {
 546         case DDI_ATTACH:
 547                 break;

 549         case DDI_RESUME:
 550                 if ((tran = ddi_get_driver_private(dip)) == NULL)
 551                         return (DDI_FAILURE);

 553                 fas = TRAN2FAS(tran);
 554                 if (!fas) {
 555                         return (DDI_FAILURE);
 556                 }
 557                 /*
 558                  * Reset hardware and softc to "no outstanding commands"
 559                  * Note that a check condition can result on first command
 560                  * to a target.
 561                  */
 562                 mutex_enter(FAS_MUTEX(fas));
 563                 fas_internal_reset(fas,
 564                     FAS_RESET_SOFTC|FAS_RESET_FAS|FAS_RESET_DMA);

 566                 (void) fas_reset_bus(fas);

 568                 fas->f_suspended = 0;

 570                 /* make sure that things get started */
 571                 (void) fas_istart(fas);
 572                 fas_check_waitQ_and_mutex_exit(fas);

 574                 mutex_enter(&fas_global_mutex);
 575                 if (fas_timeout_id == 0) {
 576                         fas_timeout_id = timeout(fas_watch, NULL, fas_tick);
 577                         fas_timeout_initted = 1;
 578                 }
```



```
 579                 mutex_exit(&fas_global_mutex);

 581                 return (DDI_SUCCESS);

 583         default:
 584                 return (DDI_FAILURE);
 585         }

 587         instance = ddi_get_instance(dip);

 589         /*
 590          * Since we know that some instantiations of this device can
 591          * be plugged into slave-only SBus slots, check to see whether
 592          * this is one such.
 593          */
 594         if (ddi_slaveonly(dip) == DDI_SUCCESS) {
 595                 cmn_err(CE_WARN,
 596                     "fas%d: device in slave-only slot", instance);
 597                 return (DDI_FAILURE);
 598         }

 600         if (ddi_intr_hilevel(dip, 0)) {
 601                 /*
 602                  * Interrupt number '0' is a high-level interrupt.
 603                  * At this point you either add a special interrupt
 604                  * handler that triggers a soft interrupt at a lower level,
 605                  * or - more simply and appropriately here - you just
 606                  * fail the attach.
 607                  */
 608                 cmn_err(CE_WARN,
 609                     "fas%d: Device is using a hilevel intr", instance);
 610                 return (DDI_FAILURE);
 611         }

 613         /*
 614          * Allocate softc information.
 615          */
 616         if (ddi_soft_state_zalloc(fas_state, instance) != DDI_SUCCESS) {
 617                 cmn_err(CE_WARN,
 618                     "fas%d: cannot allocate soft state", instance);
 619                 goto fail;
 620         }

 622         fas = (struct fas *)ddi_get_soft_state(fas_state, instance);

 624         if (fas == NULL) {
 625                 goto fail;
 626         }

 628         /*
 629          * map in device registers
 630          */
 631         dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
 632         dev_attr.devacc_attr_endian_flags = DDI_NEVERSWAP_ACC;
 633         dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

 635         if (ddi_regs_map_setup(dip, (uint_t)0, (caddr_t *)&dmar,
 636             (off_t)0, (off_t)sizeof (struct dma),
 637             &dev_attr, &fas->f_dmar_acc_handle) != DDI_SUCCESS) {
 638                 cmn_err(CE_WARN, "fas%d: cannot map dma", instance);
 639                 goto fail;
 640         }

 642         if (ddi_regs_map_setup(dip, (uint_t)1, (caddr_t *)&fasreg,
 643             (off_t)0, (off_t)sizeof (struct fasreg),
 644             &dev_attr, &fas->f_regs_acc_handle) != DDI_SUCCESS) {
```

```
645                 cmn_err(CE_WARN,
646                     "fas%d: unable to map fas366 registers", instance);
647                 goto fail;
648         }

650         fas_dma_attr = &dma_fasattr;
651         if (ddi_dma_alloc_handle(dip, fas_dma_attr,
652             DDI_DMA_SLEEP, NULL, &fas->f_dmahandle) != DDI_SUCCESS) {
653                 cmn_err(CE_WARN,
654                     "fas%d: cannot alloc dma handle", instance);
655                 goto fail;
656         }

658         /*
659          * allocate cmdarea and its dma handle
660          */
661         if (ddi_dma_mem_alloc(fas->f_dmahandle,
662             (uint_t)2*FIFOSIZE,
663             &dev_attr, DDI_DMA_CONSISTENT, DDI_DMA_SLEEP,
664             NULL, (caddr_t *)&fas->f_cmdarea, &rlen,
665             &fas->f_cmdarea_acc_handle) != DDI_SUCCESS) {
666                 cmn_err(CE_WARN,
667                     "fas%d: cannot alloc cmd area", instance);
668                 goto fail;
669         }

671         fas->f_reg = fasreg;
672         fas->f_dma = dmar;
673         fas->f_instance  = instance;

675         if (ddi_dma_addr_bind_handle(fas->f_dmahandle,
676             NULL, (caddr_t)fas->f_cmdarea,
677             rlen, DDI_DMA_RDWR|DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL,
678             &fas->f_dmacookie, &count) != DDI_DMA_MAPPED) {
679                 cmn_err(CE_WARN,
680                     "fas%d: cannot bind cmdarea", instance);
681                 goto fail;
682         }
683         bound_handle++;

685         ASSERT(count == 1);

687         /*
688          * Allocate a transport structure
689          */
690         tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

692         /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
693         scsi_size_clean(dip);           /* SCSI_SIZE_CLEAN_VERIFY ok */

695         /*
696          * initialize transport structure
697          */
698         fas->f_tran                     = tran;
699         fas->f_dev                      = dip;
700         tran->tran_hba_private          = fas;
701         tran->tran_tgt_private          = NULL;
702         tran->tran_tgt_init             = fas_scsi_tgt_init;
703         tran->tran_tgt_probe            = fas_scsi_tgt_probe;
704         tran->tran_tgt_free             = NULL;
705         tran->tran_start                = fas_scsi_start;
706         tran->tran_abort                = fas_scsi_abort;
707         tran->tran_reset                = fas_scsi_reset;
708         tran->tran_getcap               = fas_scsi_getcap;
709         tran->tran_setcap               = fas_scsi_setcap;
710         tran->tran_init_pkt             = fas_scsi_init_pkt;
```



```
711         tran->tran_destroy_pkt          = fas_scsi_destroy_pkt;
712         tran->tran_dmafree              = fas_scsi_dmafree;
713         tran->tran_sync_pkt             = fas_scsi_sync_pkt;
714         tran->tran_reset_notify         = fas_scsi_reset_notify;
715         tran->tran_get_bus_addr         = NULL;
716         tran->tran_get_name             = NULL;
717         tran->tran_quiesce              = fas_scsi_quiesce;
718         tran->tran_unquiesce            = fas_scsi_unquiesce;
719         tran->tran_bus_reset            = NULL;
720         tran->tran_add_eventcall        = NULL;
721         tran->tran_get_eventcookie      = NULL;
722         tran->tran_post_event           = NULL;
723         tran->tran_remove_eventcall     = NULL;

725         fas->f_force_async = 0;

727         /*
728          * disable tagged queuing and wide for all targets
729          * (will be enabled by target driver if required)
730          * sync is enabled by default
731          */
732         fas->f_nowide = fas->f_notag = ALL_TARGETS;
733         fas->f_force_narrow = ALL_TARGETS;

735         /*
736          * By default we assume embedded devices and save time
737          * checking for timeouts in fas_watch() by skipping
738          * the rest of luns
739          * If we're talking to any non-embedded devices,
740          * we can't cheat and skip over non-zero luns anymore
741          * in fas_watch() and fas_ustart().
742          */
743         fas->f_dslot = NLUNS_PER_TARGET;

745         /*
746          * f_active is used for saving disconnected cmds;
747          * For tagged targets, we need to increase the size later
748          * Only allocate for Lun == 0, if we probe a lun > 0 then
749          * we allocate an active structure
750          * If TQ gets enabled then we need to increase the size
751          * to hold 256 cmds
752          */
753         for (slot = 0; slot < N_SLOTS; slot += NLUNS_PER_TARGET) {
754                 (void) fas_alloc_active_slots(fas, slot, KM_SLEEP);
755         }

757         /*
758          * initialize the qfull retry counts
759          */
760         for (i = 0; i < NTARGETS_WIDE; i++) {
761                 fas->f_qfull_retries[i] = QFULL_RETRIES;
762                 fas->f_qfull_retry_interval[i] =
763                     drv_usectohz(QFULL_RETRY_INTERVAL * 1000);

765         }

767         /*
768          * Initialize throttles.
769          */
770         fas_set_throttles(fas, 0, N_SLOTS, MAX_THROTTLE);

772         /*
773          * Initialize mask of deferred property updates
774          */
775         fas->f_props_update = 0;
```

```
777         /*
778          * set host ID
779          */
780         fas->f_fasconf = DEFAULT_HOSTID;
781         id = ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0, "initiator-id", -1);
782         if (id == -1) {
783                 id = ddi_prop_get_int(DDI_DEV_T_ANY,    dip, 0,
784                     "scsi-initiator-id", -1);
785         }
786         if (id != DEFAULT_HOSTID && id >= 0 && id < NTARGETS_WIDE) {
787                 fas_log(fas, CE_NOTE, "?initiator SCSI ID now %d\n", id);
788                 fas->f_fasconf = (uchar_t)id;
789         }

791         /*
792          * find the burstsize and reduce ours if necessary
793          */
794         fas->f_dma_attr = fas_dma_attr;
795         fas->f_dma_attr->dma_attr_burstsizes &=
796             ddi_dma_burstsizes(fas->f_dmahandle);

798 #ifdef FASDEBUG
799         fas->f_dma_attr->dma_attr_burstsizes &= fas_burstsizes_limit;
800         IPRINTF1("dma burstsize=%x\n", fas->f_dma_attr->dma_attr_burstsizes);
801 #endif
802         /*
803          * Attach this instance of the hba
804          */
805         if (scsi_hba_attach_setup(dip, fas->f_dma_attr, tran, 0) !=
806             DDI_SUCCESS) {
807                 fas_log(fas, CE_WARN, "scsi_hba_attach_setup failed");
808                 goto fail;
809         }
810         hba_attached++;

812         /*
813          * if scsi-options property exists, use it
814          */
815         fas->f_scsi_options = ddi_prop_get_int(DDI_DEV_T_ANY,
816             dip, 0, "scsi-options", DEFAULT_SCSI_OPTIONS);

818         /*
819          * if scsi-selection-timeout property exists, use it
820          */
821         fas_selection_timeout = ddi_prop_get_int(DDI_DEV_T_ANY,
822             dip, 0, "scsi-selection-timeout", SCSI_DEFAULT_SELECTION_TIMEOUT);

824         /*
825          * if hm-rev property doesn't exist, use old scheme for rev
826          */
827         hm_rev = ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0,
828             "hm-rev", -1);

830         if (hm_rev == 0xa0 || hm_rev == -1) {
831                 if (DMAREV(dmar) != 0) {
832                         fas->f_hm_rev = 0x20;
833                         fas_log(fas, CE_WARN,
834                             "obsolete rev 2.0 FEPS chip, "
835                             "possible data corruption");
836                 } else {
837                         fas->f_hm_rev = 0x10;
838                         fas_log(fas, CE_WARN,
839                             "obsolete and unsupported rev 1.0 FEPS chip");
840                         goto fail;
841                 }
842         } else if (hm_rev == 0x20) {
```



```
843                 fas->f_hm_rev = 0x21;
844                 fas_log(fas, CE_WARN, "obsolete rev 2.1 FEPS chip");
845         } else {
846                 fas->f_hm_rev = (uchar_t)hm_rev;
847                 fas_log(fas, CE_NOTE, "?rev %x.%x FEPS chip\n",
848                     (hm_rev >> 4) & 0xf, hm_rev & 0xf);
849         }

851         if ((fas->f_scsi_options & SCSI_OPTIONS_SYNC) == 0) {
852                 fas->f_nosync = ALL_TARGETS;
853         }

855         if ((fas->f_scsi_options & SCSI_OPTIONS_WIDE) == 0) {
856                 fas->f_nowide = ALL_TARGETS;
857         }

859         /*
860          * if target<n>-scsi-options property exists, use it;
861          * otherwise use the f_scsi_options
862          */
863         for (i = 0; i < NTARGETS_WIDE; i++) {
864                 (void) sprintf(prop_str, prop_template, i);
865                 fas->f_target_scsi_options[i] = ddi_prop_get_int(
866                     DDI_DEV_T_ANY, dip, 0, prop_str, -1);

868                 if (fas->f_target_scsi_options[i] != -1) {
869                         fas_log(fas, CE_NOTE, "?target%x-scsi-options=0x%x\n",
870                             i, fas->f_target_scsi_options[i]);
871                         fas->f_target_scsi_options_defined |= 1 << i;
872                 } else {
873                         fas->f_target_scsi_options[i] = fas->f_scsi_options;
874                 }
875                 if (((fas->f_target_scsi_options[i] &
876                     SCSI_OPTIONS_DR) == 0) &&
877                     (fas->f_target_scsi_options[i] & SCSI_OPTIONS_TAG)) {
878                         fas->f_target_scsi_options[i] &= ~SCSI_OPTIONS_TAG;
879                         fas_log(fas, CE_WARN,
880                             "Disabled TQ since disconnects are disabled");
881                 }
882         }

884         fas->f_scsi_tag_age_limit =
885             ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0, "scsi-tag-age-limit",
886             DEFAULT_TAG_AGE_LIMIT);

888         fas->f_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
889             dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
890         if (fas->f_scsi_reset_delay == 0) {
891                 fas_log(fas, CE_NOTE,
892                     "scsi_reset_delay of 0 is not recommended,"
893                     " resetting to SCSI_DEFAULT_RESET_DELAY\n");
894                 fas->f_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
895         }

897         /*
898          * get iblock cookie and initialize mutexes
899          */
900         if (ddi_get_iblock_cookie(dip, (uint_t)0, &fas->f_iblock)
901             != DDI_SUCCESS) {
902                 cmn_err(CE_WARN, "fas_attach: cannot get iblock cookie");
903                 goto fail;
904         }

906         mutex_init(&fas->f_mutex, NULL, MUTEX_DRIVER, fas->f_iblock);
907         cv_init(&fas->f_cv, NULL, CV_DRIVER, NULL);
```

```
909         /*
910          * initialize mutex for waitQ
911          */
912         mutex_init(&fas->f_waitQ_mutex, NULL, MUTEX_DRIVER, fas->f_iblock);
913         mutex_init_done++;

915         /*
916          * initialize callback mechanism (immediate callback)
917          */
918         mutex_enter(&fas_global_mutex);
919         if (fas_init_callbacks(fas)) {
920                 mutex_exit(&fas_global_mutex);
921                 goto fail;
922         }
923         mutex_exit(&fas_global_mutex);

925         /*
926          * kstat_intr support
927          */
928         (void) sprintf(buf, "fas%d", instance);
929         fas->f_intr_kstat = kstat_create("fas", instance, buf, "controller", \
930             KSTAT_TYPE_INTR, 1, KSTAT_FLAG_PERSISTENT);
931         if (fas->f_intr_kstat)
932                 kstat_install(fas->f_intr_kstat);

934         /*
935          * install interrupt handler
936          */
937         mutex_enter(FAS_MUTEX(fas));
938         if (ddi_add_intr(dip, (uint_t)0, &fas->f_iblock, NULL,
939             fas_intr, (caddr_t)fas)) {
940                 cmn_err(CE_WARN, "fas: cannot add intr");
941                 mutex_exit(FAS_MUTEX(fas));
942                 goto fail;
943         }
944         intr_added++;

946         /*
947          * initialize fas chip
948          */
949         if (fas_init_chip(fas, id))     {
950                 cmn_err(CE_WARN, "fas: cannot initialize");
951                 mutex_exit(FAS_MUTEX(fas));
952                 goto fail;
953         }
954         mutex_exit(FAS_MUTEX(fas));

956         /*
957          * create kmem cache for packets
958          */
959         (void) sprintf(buf, "fas%d_cache", instance);
960         fas->f_kmem_cache = kmem_cache_create(buf,
961             EXTCMD_SIZE, 8,
962             fas_kmem_cache_constructor, fas_kmem_cache_destructor,
963             NULL, (void *)fas, NULL, 0);
964         if (fas->f_kmem_cache == NULL) {
965                 cmn_err(CE_WARN, "fas: cannot create kmem_cache");
966                 goto fail;
967         }

969         /*
970          * at this point, we are not going to fail the attach
971          * so there is no need to undo the rest:
972          *
973          * add this fas to the list, this makes debugging easier
974          * and fas_watch() needs it to walk thru all fas's
```



```
975          */
976         rw_enter(&fas_global_rwlock, RW_WRITER);
977         if (fas_head == NULL) {
978                 fas_head = fas;
979         } else {
980                 fas_tail->f_next = fas;
981         }
982         fas_tail = fas;         /* point to last fas in list */
983         rw_exit(&fas_global_rwlock);

985         /*
986          * there is one watchdog handler for all driver instances.
987          * start the watchdog if it hasn't been done yet
988          */
989         mutex_enter(&fas_global_mutex);
990         if (fas_scsi_watchdog_tick == 0) {
991                 fas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
992                     dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);
993                 if (fas_scsi_watchdog_tick != DEFAULT_WD_TICK) {
994                         fas_log(fas, CE_NOTE, "?scsi-watchdog-tick=%d\n",
995                             fas_scsi_watchdog_tick);
996                 }
997                 fas_tick = drv_sectohz((clock_t)fas_scsi_watchdog_tick);
997                 fas_tick = drv_usectohz((clock_t)
998                     fas_scsi_watchdog_tick * 1000000);
998                 IPRINTF2("fas scsi watchdog tick=%x, fas_tick=%lx\n",
999                     fas_scsi_watchdog_tick, fas_tick);
1000                if (fas_timeout_id == 0) {
1001                        fas_timeout_id = timeout(fas_watch, NULL, fas_tick);
1002                        fas_timeout_initted = 1;
1003                }
1004        }
1005        mutex_exit(&fas_global_mutex);

1007        ddi_report_dev(dip);

1009        return (DDI_SUCCESS);

1011 fail:
1012        cmn_err(CE_WARN, "fas%d: cannot attach", instance);
1013        if (fas) {
1014                for (slot = 0; slot < N_SLOTS; slot++) {
1015                        struct f_slots *active = fas->f_active[slot];
1016                        if (active) {
1017                                kmem_free(active, active->f_size);
1018                                fas->f_active[slot] = NULL;
1019                        }
1020                }
1021                if (mutex_init_done) {
1022                        mutex_destroy(&fas->f_mutex);
1023                        mutex_destroy(&fas->f_waitQ_mutex);
1024                        cv_destroy(&fas->f_cv);
1025                }
1026                if (intr_added) {
1027                        ddi_remove_intr(dip, (uint_t)0, fas->f_iblock);
1028                }
1029                /*
1030                 * kstat_intr support
1031                 */
1032                if (fas->f_intr_kstat) {
1033                        kstat_delete(fas->f_intr_kstat);
1034                }
1035                if (hba_attached) {
1036                        (void) scsi_hba_detach(dip);
1037                }
1038                if (tran) {
```

```
1039                        scsi_hba_tran_free(tran);
1040                }
1041                if (fas->f_kmem_cache) {
1042                        kmem_cache_destroy(fas->f_kmem_cache);
1043                }
1044                if (fas->f_cmdarea) {
1045                        if (bound_handle) {
1046                                (void) ddi_dma_unbind_handle(fas->f_dmahandle);
1047                        }
1048                        ddi_dma_mem_free(&fas->f_cmdarea_acc_handle);
1049                }
1050                if (fas->f_dmahandle) {
1051                        ddi_dma_free_handle(&fas->f_dmahandle);
1052                }
1053                fas_destroy_callbacks(fas);
1054                if (fas->f_regs_acc_handle) {
1055                        ddi_regs_map_free(&fas->f_regs_acc_handle);
1056                }
1057                if (fas->f_dmar_acc_handle) {
1058                        ddi_regs_map_free(&fas->f_dmar_acc_handle);
1059                }
1060                ddi_soft_state_free(fas_state, instance);

1062                ddi_remove_minor_node(dip, NULL);
1063        }
1064        return (DDI_FAILURE);
1065 }
```

*_____unchanged_portion_omitted_*

```
1356 static int
1357 fas_quiesce_bus(struct fas *fas)
1358 {
1359        mutex_enter(FAS_MUTEX(fas));
1360        IPRINTF("fas_quiesce: QUIESCEing\n");
1361        IPRINTF3("fas_quiesce: ncmds (%d) ndisc (%d) state (%d)\n",
1362            fas->f_ncmds, fas->f_ndisc, fas->f_softstate);
1363        fas_set_throttles(fas, 0, N_SLOTS, HOLD_THROTTLE);
1364        if (fas_check_outstanding(fas)) {
1365                fas->f_softstate |= FAS_SS_DRAINING;
1366                fas->f_quiesce_timeid = timeout(fas_ncmds_checkdrain,
1367                    fas, drv_sectohz(FAS_QUIESCE_TIMEOUT));
1368                    fas, (FAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
1368                if (cv_wait_sig(FAS_CV(fas), FAS_MUTEX(fas)) == 0) {
1369                        /*
1370                         * quiesce has been interrupted.
1371                         */
1372                        IPRINTF("fas_quiesce: abort QUIESCE\n");
1373                        fas->f_softstate &= ~FAS_SS_DRAINING;
1374                        fas_set_throttles(fas, 0, N_SLOTS, MAX_THROTTLE);
1375                        (void) fas_istart(fas);
1376                        if (fas->f_quiesce_timeid != 0) {
1377                                mutex_exit(FAS_MUTEX(fas));
1378 #ifndef __lock_lint     /* warlock complains but there is a NOTE on this */
1379                                (void) untimeout(fas->f_quiesce_timeid);
1380                                fas->f_quiesce_timeid = 0;
1381 #endif
1382                                return (-1);
1383                        }
1384                        mutex_exit(FAS_MUTEX(fas));
1385                        return (-1);
1386                } else {
1387                        IPRINTF("fas_quiesce: bus is QUIESCED\n");
1388                        ASSERT(fas->f_quiesce_timeid == 0);
1389                        fas->f_softstate &= ~FAS_SS_DRAINING;
1390                        fas->f_softstate |= FAS_SS_QUIESCED;
1391                        mutex_exit(FAS_MUTEX(fas));
```



```
1392                        return (0);
1393                }
1394        }
1395        IPRINTF("fas_quiesce: bus was not busy QUIESCED\n");
1396        mutex_exit(FAS_MUTEX(fas));
1397        return (0);
1398 }
```

*_____unchanged_portion_omitted_*

```
1413 /*
1414  * invoked from timeout() to check the number of outstanding commands
1415  */
1416 static void
1417 fas_ncmds_checkdrain(void *arg)
1418 {
1419        struct fas *fas = arg;

1421        mutex_enter(FAS_MUTEX(fas));
1422        IPRINTF3("fas_checkdrain: ncmds (%d) ndisc (%d) state (%d)\n",
1423            fas->f_ncmds, fas->f_ndisc, fas->f_softstate);
1424        if (fas->f_softstate & FAS_SS_DRAINING) {
1425                fas->f_quiesce_timeid = 0;
1426                if (fas_check_outstanding(fas) == 0) {
1427                        IPRINTF("fas_drain: bus has drained\n");
1428                        cv_signal(FAS_CV(fas));
1429                } else {
1430                        /*
1431                         * throttle may have been reset by a bus reset
1432                         * or fas_runpoll()
1433                         * XXX shouldn't be necessary
1434                         */
1435                        fas_set_throttles(fas, 0, N_SLOTS, HOLD_THROTTLE);
1436                        IPRINTF("fas_drain: rescheduling timeout\n");
1437                        fas->f_quiesce_timeid = timeout(fas_ncmds_checkdrain,
1438                            fas, drv_sectohz(FAS_QUIESCE_TIMEOUT));
1439                            fas, (FAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
1439                }
1440        }
1441        mutex_exit(FAS_MUTEX(fas));
1442 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  194641 Wed Aug 19 07:25:27 2015
new/usr/src/uts/sun/io/scsi/adapters/sf.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 440 /*
 441  * either attach or resume this driver
 442  */
 443 static int
 444 sf_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 445 {
 446         int instance;
 447         int mutex_initted = FALSE;
 448         uint_t ccount;
 449         size_t i, real_size;
 450         struct fcal_transport *handle;
 451         char buf[64];
 452         struct sf *sf, *tsf;
 453         scsi_hba_tran_t *tran = NULL;
 454         int     handle_bound = FALSE;
 455         kthread_t *tp;


 458         switch ((int)cmd) {

 460         case DDI_RESUME:

 462                 /*
 463                  * we've previously been SF_STATE_OFFLINEd by a DDI_SUSPEND,
 464                  * so time to undo that and get going again by forcing a
 465                  * lip
 466                  */

 468                 instance = ddi_get_instance(dip);

 470                 sf = ddi_get_soft_state(sf_state, instance);
 471                 SF_DEBUG(2, (sf, CE_CONT,
 472                     "sf_attach: DDI_RESUME for sf%d\n", instance));
 473                 if (sf == NULL) {
 474                         cmn_err(CE_WARN, "sf%d: bad soft state", instance);
 475                         return (DDI_FAILURE);
 476                 }

 478                 /*
 479                  * clear suspended flag so that normal operations can resume
 480                  */
 481                 mutex_enter(&sf->sf_mutex);
 482                 sf->sf_state &= ~SF_STATE_SUSPENDED;
 483                 mutex_exit(&sf->sf_mutex);

 485                 /*
 486                  * force a login by setting our state to offline
 487                  */
 488                 sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
 489                 sf->sf_state = SF_STATE_OFFLINE;

 491                 /*
 492                  * call transport routine to register state change and
 493                  * ELS callback routines (to register us as a ULP)
 494                  */
 495                 soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
 496                     sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP,
 497                     sf_statec_callback, sf_unsol_els_callback, NULL, sf);
```



```
 499                 /*
 500                  * call transport routine to force loop initialization
 501                  */
 502                 (void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
 503                     sf->sf_sochandle->fcal_portno, 0, FCAL_NO_LIP);

 505                 /*
 506                  * increment watchdog init flag, setting watchdog timeout
 507                  * if we are the first (since somebody has to do it)
 508                  */
 509                 mutex_enter(&sf_global_mutex);
 510                 if (!sf_watchdog_init++) {
 511                         mutex_exit(&sf_global_mutex);
 512                         sf_watchdog_id = timeout(sf_watch,
 513                             (caddr_t)0, sf_watchdog_tick);
 514                 } else {
 515                         mutex_exit(&sf_global_mutex);
 516                 }

 518                 return (DDI_SUCCESS);

 520         case DDI_ATTACH:

 522                 /*
 523                  * this instance attaching for the first time
 524                  */

 526                 instance = ddi_get_instance(dip);

 528                 if (ddi_soft_state_zalloc(sf_state, instance) !=
 529                     DDI_SUCCESS) {
 530                         cmn_err(CE_WARN, "sf%d: failed to allocate soft state",
 531                             instance);
 532                         return (DDI_FAILURE);
 533                 }

 535                 sf = ddi_get_soft_state(sf_state, instance);
 536                 SF_DEBUG(4, (sf, CE_CONT,
 537                     "sf_attach: DDI_ATTACH for sf%d\n", instance));
 538                 if (sf == NULL) {
 539                         /* this shouldn't happen since we just allocated it */
 540                         cmn_err(CE_WARN, "sf%d: bad soft state", instance);
 541                         return (DDI_FAILURE);
 542                 }

 544                 /*
 545                  * from this point on, if there's an error, we must de-allocate
 546                  * soft state before returning DDI_FAILURE
 547                  */

 549                 if ((handle = ddi_get_parent_data(dip)) == NULL) {
 550                         cmn_err(CE_WARN,
 551                             "sf%d: failed to obtain transport handle",
 552                             instance);
 553                         goto fail;
 554                 }

 556                 /* fill in our soft state structure */
 557                 sf->sf_dip = dip;
 558                 sf->sf_state = SF_STATE_INIT;
 559                 sf->sf_throttle = handle->fcal_cmdmax;
 560                 sf->sf_sochandle = handle;
 561                 sf->sf_socp = handle->fcal_handle;
 562                 sf->sf_check_n_close = 0;

 564                 /* create a command/response buffer pool for this instance */
```

```
565                 if (sf_add_cr_pool(sf) != DDI_SUCCESS) {
566                         cmn_err(CE_WARN,
567                             "sf%d: failed to allocate command/response pool",
568                             instance);
569                         goto fail;
570                 }

572                 /* create a a cache for this instance */
573                 (void) sprintf(buf, "sf%d_cache", instance);
574                 sf->sf_pkt_cache = kmem_cache_create(buf,
575                     sizeof (fcal_packet_t) + sizeof (struct sf_pkt) +
576                     scsi_pkt_size(), 8,
577                     sf_kmem_cache_constructor, sf_kmem_cache_destructor,
578                     NULL, NULL, NULL, 0);
579                 if (sf->sf_pkt_cache == NULL) {
580                         cmn_err(CE_WARN, "sf%d: failed to allocate kmem cache",
581                             instance);
582                         goto fail;
583                 }

585                 /* set up a handle and allocate memory for DMA */
586                 if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->
587                     fcal_dmaattr, DDI_DMA_DONTWAIT, NULL, &sf->
588                     sf_lilp_dmahandle) != DDI_SUCCESS) {
589                         cmn_err(CE_WARN,
590                             "sf%d: failed to allocate dma handle for lilp map",
591                             instance);
592                         goto fail;
593                 }
594                 i = sizeof (struct fcal_lilp_map) + 1;
595                 if (ddi_dma_mem_alloc(sf->sf_lilp_dmahandle,
596                     i, sf->sf_sochandle->
597                     fcal_accattr, DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
598                     (caddr_t *)&sf->sf_lilp_map, &real_size,
599                     &sf->sf_lilp_acchandle) != DDI_SUCCESS) {
600                         cmn_err(CE_WARN, "sf%d: failed to allocate lilp map",
601                             instance);
602                         goto fail;
603                 }
604                 if (real_size < i) {
605                         /* no error message ??? */
606                         goto fail;              /* trouble allocating memory */
607                 }

609                 /*
610                  * set up the address for the DMA transfers (getting a cookie)
611                  */
612                 if (ddi_dma_addr_bind_handle(sf->sf_lilp_dmahandle, NULL,
613                     (caddr_t)sf->sf_lilp_map, real_size,
614                     DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
615                     &sf->sf_lilp_dmacookie, &ccount) != DDI_DMA_MAPPED) {
616                         cmn_err(CE_WARN,
617                             "sf%d: failed to bind dma handle for lilp map",
618                             instance);
619                         goto fail;
620                 }
621                 handle_bound = TRUE;
622                 /* ensure only one cookie was allocated */
623                 if (ccount != 1) {
624                         goto fail;
625                 }

627                 /* ensure LILP map and DMA cookie addresses are even?? */
628                 sf->sf_lilp_map = (struct fcal_lilp_map *)(((uintptr_t)sf->
629                     sf_lilp_map + 1) & ~1);
630                 sf->sf_lilp_dmacookie.dmac_address = (sf->
```



```
631                     sf_lilp_dmacookie.dmac_address + 1) & ~1;

633                 /* set up all of our mutexes and condition variables */
634                 mutex_init(&sf->sf_mutex, NULL, MUTEX_DRIVER, NULL);
635                 mutex_init(&sf->sf_cmd_mutex, NULL, MUTEX_DRIVER, NULL);
636                 mutex_init(&sf->sf_cr_mutex, NULL, MUTEX_DRIVER, NULL);
637                 mutex_init(&sf->sf_hp_daemon_mutex, NULL, MUTEX_DRIVER, NULL);
638                 cv_init(&sf->sf_cr_cv, NULL, CV_DRIVER, NULL);
639                 cv_init(&sf->sf_hp_daemon_cv, NULL, CV_DRIVER, NULL);

641                 mutex_initted = TRUE;

643                 /* create our devctl minor node */
644                 if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
645                     SF_INST2DEVCTL_MINOR(instance),
646                     DDI_NT_NEXUS, 0) != DDI_SUCCESS) {
647                         cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
648                             " for devctl", instance);
649                         goto fail;
650                 }

652                 /* create fc minor node */
653                 if (ddi_create_minor_node(dip, "fc", S_IFCHR,
654                     SF_INST2FC_MINOR(instance), DDI_NT_FC_ATTACHMENT_POINT,
655                     0) != DDI_SUCCESS) {
656                         cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
657                             " for fc", instance);
658                         goto fail;
659                 }
660                 /* allocate a SCSI transport structure */
661                 tran = scsi_hba_tran_alloc(dip, 0);
662                 if (tran == NULL) {
663                         /* remove all minor nodes created */
664                         ddi_remove_minor_node(dip, NULL);
665                         cmn_err(CE_WARN, "sf%d: scsi_hba_tran_alloc failed",
666                             instance);
667                         goto fail;
668                 }

670                 /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
671                 scsi_size_clean(dip);           /* SCSI_SIZE_CLEAN_VERIFY ok */

673                 /* save ptr to new transport structure and fill it in */
674                 sf->sf_tran = tran;

676                 tran->tran_hba_private          = sf;
677                 tran->tran_tgt_private          = NULL;
678                 tran->tran_tgt_init             = sf_scsi_tgt_init;
679                 tran->tran_tgt_probe            = NULL;
680                 tran->tran_tgt_free             = sf_scsi_tgt_free;

682                 tran->tran_start                = sf_start;
683                 tran->tran_abort                = sf_abort;
684                 tran->tran_reset                = sf_reset;
685                 tran->tran_getcap               = sf_getcap;
686                 tran->tran_setcap               = sf_setcap;
687                 tran->tran_init_pkt             = sf_scsi_init_pkt;
688                 tran->tran_destroy_pkt          = sf_scsi_destroy_pkt;
689                 tran->tran_dmafree              = sf_scsi_dmafree;
690                 tran->tran_sync_pkt             = sf_scsi_sync_pkt;
691                 tran->tran_reset_notify         = sf_scsi_reset_notify;

693                 /*
694                  * register event notification routines with scsa
695                  */
696                 tran->tran_get_eventcookie      = sf_bus_get_eventcookie;
```

```
697                 tran->tran_add_eventcall        = sf_bus_add_eventcall;
698                 tran->tran_remove_eventcall     = sf_bus_remove_eventcall;
699                 tran->tran_post_event           = sf_bus_post_event;

701                 /*
702                  * register bus configure/unconfigure
703                  */
704                 tran->tran_bus_config           = sf_scsi_bus_config;
705                 tran->tran_bus_unconfig         = sf_scsi_bus_unconfig;

707                 /*
708                  * allocate an ndi event handle
709                  */
710                 sf->sf_event_defs = (ndi_event_definition_t *)
711                     kmem_zalloc(sizeof (sf_event_defs), KM_SLEEP);

713                 bcopy(sf_event_defs, sf->sf_event_defs,
714                     sizeof (sf_event_defs));

716                 (void) ndi_event_alloc_hdl(dip, NULL,
717                     &sf->sf_event_hdl, NDI_SLEEP);

719                 sf->sf_events.ndi_events_version = NDI_EVENTS_REV1;
720                 sf->sf_events.ndi_n_events = SF_N_NDI_EVENTS;
721                 sf->sf_events.ndi_event_defs = sf->sf_event_defs;

723                 if (ndi_event_bind_set(sf->sf_event_hdl,
724                     &sf->sf_events, NDI_SLEEP) != NDI_SUCCESS) {
725                         goto fail;
726                 }

728                 tran->tran_get_name             = sf_scsi_get_name;
729                 tran->tran_get_bus_addr         = sf_scsi_get_bus_addr;

731                 /* setup and attach SCSI hba transport */
732                 if (scsi_hba_attach_setup(dip, sf->sf_sochandle->
733                     fcal_dmaattr, tran, SCSI_HBA_TRAN_CLONE) != DDI_SUCCESS) {
734                         cmn_err(CE_WARN, "sf%d: scsi_hba_attach_setup failed",
735                             instance);
736                         goto fail;
737                 }

739                 /* set up kstats */
740                 if ((sf->sf_ksp = kstat_create("sf", instance, "statistics",
741                     "controller", KSTAT_TYPE_RAW, sizeof (struct sf_stats),
742                     KSTAT_FLAG_VIRTUAL)) == NULL) {
743                         cmn_err(CE_WARN, "sf%d: failed to create kstat",
744                             instance);
745                 } else {
746                         sf->sf_stats.version = 2;
747                         (void) sprintf(sf->sf_stats.drvr_name,
748                         "%s: %s", SF_NAME, sf_version);
749                         sf->sf_ksp->ks_data = (void *)&sf->sf_stats;
750                         sf->sf_ksp->ks_private = sf;
751                         sf->sf_ksp->ks_update = sf_kstat_update;
752                         kstat_install(sf->sf_ksp);
753                 }

755                 /* create the hotplug thread */
756                 mutex_enter(&sf->sf_hp_daemon_mutex);
757                 tp = thread_create(NULL, 0,
758                     (void (*)())sf_hp_daemon, sf, 0, &p0, TS_RUN, minclsyspri);
759                 sf->sf_hp_tid = tp->t_did;
760                 mutex_exit(&sf->sf_hp_daemon_mutex);

762                 /* add this soft state instance to the head of the list */
```



```
763                 mutex_enter(&sf_global_mutex);
764                 sf->sf_next = sf_head;
765                 tsf = sf_head;
766                 sf_head = sf;

768                 /*
769                  * find entry in list that has the same FC-AL handle (if any)
770                  */
771                 while (tsf != NULL) {
772                         if (tsf->sf_socp == sf->sf_socp) {
773                                 break;          /* found matching entry */
774                         }
775                         tsf = tsf->sf_next;
776                 }

778                 if (tsf != NULL) {
779                         /* if we found a matching entry keep track of it */
780                         sf->sf_sibling = tsf;
781                 }

783                 /*
784                  * increment watchdog init flag, setting watchdog timeout
785                  * if we are the first (since somebody has to do it)
786                  */
787                 if (!sf_watchdog_init++) {
788                         mutex_exit(&sf_global_mutex);
789                         sf_watchdog_tick = drv_sectohz(sf_watchdog_timeout);
789                         sf_watchdog_tick = sf_watchdog_timeout *
790                             drv_usectohz(1000000);
790                         sf_watchdog_id = timeout(sf_watch,
791                             NULL, sf_watchdog_tick);
792                 } else {
793                         mutex_exit(&sf_global_mutex);
794                 }

796                 if (tsf != NULL) {
797                         /*
798                          * set up matching entry to be our sibling
799                          */
800                         mutex_enter(&tsf->sf_mutex);
801                         tsf->sf_sibling = sf;
802                         mutex_exit(&tsf->sf_mutex);
803                 }

805                 /*
806                  * create this property so that PM code knows we want
807                  * to be suspended at PM time
808                  */
809                 (void) ddi_prop_update_string(DDI_DEV_T_NONE, dip,
810                     PM_HARDWARE_STATE_PROP, PM_NEEDS_SUSPEND_RESUME);

812                 /* log the fact that we have a new device */
813                 ddi_report_dev(dip);

815                 /*
816                  * force a login by setting our state to offline
817                  */
818                 sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
819                 sf->sf_state = SF_STATE_OFFLINE;

821                 /*
822                  * call transport routine to register state change and
823                  * ELS callback routines (to register us as a ULP)
824                  */
825                 soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
826                     sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP,
```

```
827                     sf_statec_callback, sf_unsol_els_callback, NULL, sf);

829                 /*
830                  * call transport routine to force loop initialization
831                  */
832                 (void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
833                     sf->sf_sochandle->fcal_portno, 0, FCAL_NO_LIP);
834                 sf->sf_reset_time = ddi_get_lbolt64();
835                 return (DDI_SUCCESS);

837         default:
838                 return (DDI_FAILURE);
839         }

841 fail:
842         cmn_err(CE_WARN, "sf%d: failed to attach", instance);

844         /*
845          * Unbind and free event set
846          */
847         if (sf->sf_event_hdl) {
848                 (void) ndi_event_unbind_set(sf->sf_event_hdl,
849                     &sf->sf_events, NDI_SLEEP);
850                 (void) ndi_event_free_hdl(sf->sf_event_hdl);
851         }

853         if (sf->sf_event_defs) {
854                 kmem_free(sf->sf_event_defs, sizeof (sf_event_defs));
855         }

857         if (sf->sf_tran != NULL) {
858                 scsi_hba_tran_free(sf->sf_tran);
859         }
860         while (sf->sf_cr_pool != NULL) {
861                 sf_crpool_free(sf);
862         }
863         if (sf->sf_lilp_dmahandle != NULL) {
864                 if (handle_bound) {
865                         (void) ddi_dma_unbind_handle(sf->sf_lilp_dmahandle);
866                 }
867                 ddi_dma_free_handle(&sf->sf_lilp_dmahandle);
868         }
869         if (sf->sf_pkt_cache != NULL) {
870                 kmem_cache_destroy(sf->sf_pkt_cache);
871         }
872         if (sf->sf_lilp_map != NULL) {
873                 ddi_dma_mem_free(&sf->sf_lilp_acchandle);
874         }
875         if (sf->sf_ksp != NULL) {
876                 kstat_delete(sf->sf_ksp);
877         }
878         if (mutex_initted) {
879                 mutex_destroy(&sf->sf_mutex);
880                 mutex_destroy(&sf->sf_cmd_mutex);
881                 mutex_destroy(&sf->sf_cr_mutex);
882                 mutex_destroy(&sf->sf_hp_daemon_mutex);
883                 cv_destroy(&sf->sf_cr_cv);
884                 cv_destroy(&sf->sf_hp_daemon_cv);
885         }
886         mutex_enter(&sf_global_mutex);

888         /*
889          * kill off the watchdog if we are the last instance
890          */
891         if (!--sf_watchdog_init) {
892                 timeout_id_t tid = sf_watchdog_id;
```



```
893                 mutex_exit(&sf_global_mutex);
894                 (void) untimeout(tid);
895         } else {
896                 mutex_exit(&sf_global_mutex);
897         }

899         ddi_soft_state_free(sf_state, instance);

901         if (tran != NULL) {
902                 /* remove all minor nodes */
903                 ddi_remove_minor_node(dip, NULL);
904         }

906         return (DDI_FAILURE);
907 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11309 Wed Aug 19 07:25:28 2015
new/usr/src/uts/sun/sys/dada/targets/daddef.h
XXXX introduce drv_sectohz
**********************************************************
_____unchanged_portion_omitted_
 181 #define DCD_MAX_XFER_SIZE       (1 * 512)

 183 _NOTE(MUTEX_PROTECTS_DATA(dcd_device::dcd_mutex, dcd_disk))
 184 _NOTE(READ_ONLY_DATA(dcd_disk::un_dcd))
 185 _NOTE(READ_ONLY_DATA(dcd_disk::un_cmd_stat_size))
 186 _NOTE(SCHEME_PROTECTS_DATA("Save Sharing",
 187         dcd_disk::un_state
 188         dcd_disk::un_dklbhandle
 189         dcd_disk::un_format_in_progress))

 191 _NOTE(SCHEME_PROTECTS_DATA("stable data",
 192         dcd_disk::un_max_xfer_size
 193         dcd_disk::un_secdiv
 194         dcd_disk::un_secsize
 195         dcd_disk::un_cmd_flags
 196         dcd_disk::un_cmd_stat_size))

 198 _NOTE(SCHEME_PROTECTS_DATA("cv",
 199         dcd_disk::un_sbufp
 200         dcd_disk::un_srqbufp
 201         dcd_disk::un_sbuf_busy))

 203 _NOTE(SCHEME_PROTECTS_DATA("Unshared data",
 204         dk_cinfo
 205         uio
 206         buf
 207         dcd_pkt
 208         udcd_cmd
 209         dcd_capacity
 210         dcd_cmd
 211         dk_label
 212         dk_map32))

 214 _NOTE(SCHEME_PROTECTS_DATA("stable data", dcd_device))
 215 _NOTE(SCHEME_PROTECTS_DATA("unique per pkt", dcd_cmd))

 217 #endif  /* defined(_KERNEL) || defined(_KMEMUSER) */


 220 /*
 221  * Disk driver states
 222  */

 224 #define DCD_STATE_NORMAL        0
 225 #define DCD_STATE_OFFLINE       1
 226 #define DCD_STATE_RWAIT         2
 227 #define DCD_STATE_DUMPING       3
 228 #define DCD_STATE_SUSPENDED     4
 229 #define DCD_STATE_FATAL         5
 230 #define DCD_STATE_PM_SUSPENDED  6

 232 /*
 233  * Disk power levels.
 234  */
 235 #define DCD_DEVICE_ACTIVE       0x2
 236 #define DCD_DEVICE_IDLE         0x1
 237 #define DCD_DEVICE_STANDBY      0x0

 239 /*
 240  * Macros used in obtaining the device ID for the disk.
```



```
 241  */
 242 #define DCD_SERIAL_NUMBER_LENGTH        20
 243 #define DCD_MODEL_NUMBER_LENGTH         40

 245 /*
 246  * The table is to be interpreted as follows: The rows lists all the states
 247  * and each column is a state that a state in each row *can* reach. The entries
 248  * in the table list the event that cause that transition to take place.
 249  * For e.g.: To go from state RWAIT to SUSPENDED, event (d)-- which is the
 250  * invocation of DDI_SUSPEND-- has to take place. Note the same event could
 251  * cause the transition from one state to two different states. e.g., from
 252  * state SUSPENDED, when we get a DDI_RESUME, we just go back to the *last
 253  * state* whatever that might be. (NORMAL or OFFLINE).
 254  *
 255  *
 256  * State Transition Table:
 257  *
 258  *                      NORMAL  OFFLINE  RWAIT  DUMPING  SUSPENDED
 259  *
 260  *      NORMAL          -       (a)     (b)     (c)     (d)
 261  *
 262  *      OFFLINE         (e)     -       (e)     (c)     (d)
 263  *
 264  *      RWAIT           (f)     NP      -       (c)     (d)
 265  *
 266  *      DUMPING         NP      NP      NP      -       NP
 267  *
 268  *      SUSPENDED       (g)     (g)     (b)     NP*     -
 269  *
 270  *
 271  *      NP:     Not Possible.
 272  *      (a):    Disk does not respond.
 273  *      (b):    Packet Allocation Fails
 274  *      (c):    Panic - Crash dump
 275  *      (d):    DDI_SUSPEND is called.
 276  *      (e):    Disk has a successful I/O completed.
 277  *      (f):    sdrunout() calls sdstart() which sets it NORMAL
 278  *      (g):    DDI_RESUME is called.
 279  *      * :     When suspended, we dont change state during panic dump
 280  */


 283 /*
 284  * Error levels
 285  */

 287 #define DCDERR_ALL              0
 288 #define DCDERR_UNKNOWN          1
 289 #define DCDERR_INFORMATIONAL    2
 290 #define DCDERR_RECOVERED        3
 291 #define DCDERR_RETRYABLE        4
 292 #define DCDERR_FATAL            5

 294 /*
 295  * Parameters
 296  */

 298 /*
 299  * 60 seconds is a *very* reasonable amount of time for most slow CD
 300  * operations.
 301  */

 303 #define DCD_IO_TIME     60

 305 /*
 306  * Timeout value for ATA_FLUSH_CACHE used in DKIOCFLUSHWRITECACHE
```

```
 */
#define DCD_FLUSH_TIME  60

/*
 * 2 hours is an excessively reasonable amount of time for format operations.
 */

#define DCD_FMT_TIME    120*60

/*
 * 5 seconds is what we'll wait if we get a Busy Status back
 */

#define DCD_BSY_TIMEOUT         drv_sectohz(5)
#define DCD_BSY_TIMEOUT         (drv_usectohz(5 * 1000000))

/*
 * Number of times we'll retry a normal operation.
 *
 * This includes retries due to transport failure
 * (need to distinguish between Target and Transport failure)
 */

#define DCD_RETRY_COUNT         5


/*
 * Maximum number of units we can support
 * (controlled by room in minor device byte)
 * XXX: this is out of date!
 */
#define DCD_MAXUNIT             32

/*
 * 30 seconds is what we will wait for the IO to finish
 * before we fail the DDI_SUSPEND
 */
#define DCD_WAIT_CMDS_COMPLETE  30

/*
 * dcdintr action codes
 */

#define COMMAND_DONE            0
#define COMMAND_DONE_ERROR      1
#define QUE_COMMAND             2
#define QUE_SENSE               3
#define JUST_RETURN             4

/*
 * Indicator for Soft and hard errors
 */
#define COMMAND_SOFT_ERROR      1
#define COMMAND_HARD_ERROR      2

/*
 * Drive Types (and characteristics)
 */
#define VIDMAX 8
#define PIDMAX 16

struct dcd_drivetype {
        char    *name;          /* for debug purposes */
        char    ctype;          /* controller type */
        char    options;        /* drive options */
        ushort_t block_factor;  /* Block mode factor */
```



```
        char    pio_mode;       /* This the Pio mode number */
        char    dma_mode;       /* Multi word dma mode */
};
______unchanged_portion_omitted_
```

```
**********************************************************
    9761 Wed Aug 19 07:25:28 2015
new/usr/src/uts/sun/sys/fdreg.h
XXXX introduce drv_sectohz
**********************************************************
_____unchanged_portion_omitted_
  60 #endif  /* !_ASM */

  62 /* DSR - data rate select register */
  63 #define SWR             0x80    /* software reset */
  64 #define PD              0x40    /* power down */
  65 #define EPL             0x20    /* enable phase lock loop */
  66 #define PRECOMPMSK      0x1c    /* precomp mask */
  67 #define DRSELMSK        0x3     /* data rate select mask */

  69 /* MSR - main status register */
  70 #define RQM 0x80        /* request for master - chip needs attention */
  71 #define DIO 0x40        /* data in/out - 1 = remove bytes from fifo */
  72 #define NDM 0x20        /* non-dma mode - 1 during execution phase */
  73 #define CB  0x10        /* controller busy - command in progress */

  75 /* command types */
  76 #define GPLN 0x1b       /* gap length for read/write command */
  77 #define GPLF 0x54       /* gap length for format command */
  78 #define FDATA 0xe5      /* fill data fields during format */

  80 /* commands */

  82 /* 0x00-0x01 not defined */
  83 #define RDTRK           0x02
  84 #define SPECIFY         0x03
  85 #define SNSDSTAT        0x04
  86 #define WRTCMD          0x05
  87 #define RDCMD           0x06
  88 #define RECALIBRATE     0x07
  89 #define SNSISTAT        0x08    /* Sense Interrupt Status */
  90 #define WRTDEL          0x09    /* Write Deleted Data Sector */
  91 #define RDID            0x0A    /* Read Identifier */
  92 #define MTONOFF         0x0B    /* motor on/off */
  93 #define RDDEL           0x0C    /* Read Deleted Data Sector */
  94 #define FMTTRK          0x0D    /* Format Track */
  95 #define DUMPREG         0x0E    /* Dump Registers */
  96 #define SEEK            0x0F    /* Seek */
  97 /* 0x10-0x12 not defined */
  98 #define CONFIGURE       0x13
  99 /* 0x14-0x1F not defined */

 101 /* Modifier bits for the command byte */
 102 #define MT              0x80
 103 #define MFM             0x40
 104 #define SK              0x20
 105 #define MOT             0x80
 106 #define IPS             0x80    /* Used for South Bridge superI/O */


 109 #define SSSDTL          0xff    /* special sector size */

 111 #define NCBRW           0x09    /* number cmd bytes for read/write cmds */
 112 #define NRBRW           0x07    /* number result bytes for read/write cmds */

 114 /* results */
 115 /* status reg0 */
 116 #define IC_SR0          0xc0    /* interrupt code */
 117 #define SE_SR0          0x20    /* seek end */
 118 #define EC_SR0          0x10    /* equipment check */
 119 #define NR_SR0          0x08    /* not ready */
```



```
 120 #define H_SR0           0x04    /* head address */
 121 #define DS_SR0          0x03    /* drive select */

 123 /* status reg1 */
 124 #define EN_SR1          0x80    /* end of cylinder */
 125 #define DE_SR1          0x20    /* data error */
 126 #define OR_SR1          0x10    /* overrun/underrun */
 127 #define ND_SR1          0x04    /* no data */
 128 #define NW_SR1          0x02    /* not writable */
 129 #define MA_SR1          0x01    /* missing address mark */
 130 #define TO_SR1          0x08    /* Timeout */

 132 /* status reg3 */
 133 #define WP_SR3          0x40    /* write protected */
 134 #define T0_SR3          0x10    /* track zero */

 136 /* DOR - Digital Output register - 82077 only */
 137 #define EJECT           0x80    /* eject diskette - was in Auxio */
 138 #define EJECT_DMA       0x20    /* eject diskette - on DMA platform */
 139 #define MOTEN(unit)     (unit ? 0x30 : 0x10)    /* motor enable bit */
 140 #define DMAGATE         0x8     /* must be high to enable interrupts */
 141 #define RESET           0x4     /* reset bit */
 142 #define DRVSEL          0x1     /* drive select */

 144 /* DIR - Digital Input register - 82077 only */
 145 #define DSKCHG  0x80            /* diskette was changed - was in Auxio */

 147 #define DRV_MASK        0x03    /* drive mask for the second command byte */

 149 #ifndef _ASM
 150 #define Moton_delay     (drv_usectohz(750000))          /* motor on delay */
 151                                                         /* 0.75 seconds */
 152 #define Motoff_delay    drv_sectohz(6)          /* motor off delay */
 152 #define Motoff_delay    (6 * drv_usectohz(1000000))     /* motor off delay */
 153                                                         /* 6 seconds */

 155 /* Macros to set and retrieve data from the controller registers */
 156 #define Msr(fdc)        ddi_get8(fdc->c_handlep_cont, \
 157                                  ((uint8_t *)fdc->c_control))
 158 #define Dsr(fdc, val)   ddi_put8(fdc->c_handlep_cont, \
 159                                  ((uint8_t *)fdc->c_control),\
 160                                  ((uint8_t)val))
 161 #define Dir(fdc)        ddi_get8(fdc->c_handlep_cont, \
 162                                  ((uint8_t *)fdc->c_dir))
 163 #define Fifo(fdc)       ddi_get8(fdc->c_handlep_cont, \
 164                                  ((uint8_t *)fdc->c_fifo))
 165 #define Set_Fifo(fdc, val) ddi_put8(fdc->c_handlep_cont, \
 166                                  ((uint8_t *)fdc->c_fifo), \
 167                                  ((uint8_t)val))
 168 #define Dor(fdc)        ddi_get8(fdc->c_handlep_cont, ((uint8_t *)fdc->c_dor))
 169 #define Set_dor(fdc, val, flag) \
 170         { if (flag) \
 171                 ddi_put8(fdc->c_handlep_cont, ((uint8_t *)fdc->c_dor), \
 172                         ((uint8_t)(Dor(fdc) | (val)))); \
 173             else \
 174                 ddi_put8(fdc->c_handlep_cont, ((uint8_t *)fdc->c_dor), \
 175                         ((uint8_t)(Dor(fdc) & ~(val)))); }
 176 #endif  /* !_ASM */

 178 /*
 179  * Auxio Registers
 180  */

 182 /*
 183  * Definitions and structures for the floppy Auxiliary Input/Output register
 184  * for the muchio, slavio, and cheerio I/O subsystem chips
```

```
185  *
186  * In general, muchio is found on sun4c, slavio is found on sun4m and sun4u
187  * with Sbus.  Cheerio is found on sun4u with a PCI bus.
188  *
189  *
190  *
191  *                      07   06   05   04   03   02   01   00
192  *      muchio          1    1    DEN  CHG  SEL  TC   EJCT LED
193  *      slavio          1    1    DEN  0    IMUX 0    TC   LED
194  *
195  * The auxio register is designed poorly from a software perspective.
196  *  a) it supports other functions as well as floppy
197  *  b) TC is at a different bit position for muchio versus sun4m
198  *
199  * The cheerio auxio register is only for the floppy and it is a 32 bit
200  * register.  It does not contain a TC because the cheerio supports
201  * floppy DMA.  Please note that on the slavio auxio, the Digital
202  * Output register of the floppy controller contains a Density Select bit.
203  * On the cheerio, this bit is muxed with another
204  * signal.  So, the cheerio auxio register contains a density select bit.
205  *
206  *    cheerio auxio bit name    bit#
207  *    ------------------------------
208  *    Floppy density sense      0
209  *    Floppy desnity select     1
210  *    Unused                    31:1
211  *
212  */

214 /*
215  * muchio/slavio: Bits of the auxio register
216  *      - when writing to the auxio register, the bits represented by
217  *        AUX_MBO and AUX_MBO4M must be one
218  */

220 #define AUX_MBO         0xF0            /* Must be written with ones */
221 #define AUX_MBO4M       0xC0            /* Must be written with ones */

223 #define AUX_TC4M        0x02    /* 4m Floppy termnal count */
224                                 /* 1 = transfer over */
225 #define AUX_TC          0x04    /* 4c Floppy terminal count */
226                                 /* 1 = transfer over */
227 #define AUX_DENSITY     0x20    /* Floppy density (input value) */
228                                 /* 1 = high, 0 = low */


231 /*
232  * muchio additional floppy auxio bits
233  * slavio uses internal dor for these bits
234  */

236 #define AUX_DISKCHG     0x10            /* Floppy diskette change (input) */
237                                         /* 1 = new diskette inserted */
238 #define AUX_DRVSELECT   0x08            /* Floppy drive select (output) */
239                                         /* 1 = selected, 0 = deselected */
240 #define AUX_EJECT       0x02            /* Floppy eject (output,NON inverted) */
241                                         /* 0 = eject the diskette */
242 /*
243  * cheerio additional floppy auxio bits
244  */

246 #define AUX_MEDIUM_DENSITY      0x0     /* Use medium density */
247 #define AUX_HIGH_DENSITY        0x2

249 /*
250  * macros to set the Cheerio auxio registers.
```



```
251  */

253 #define Set_auxio(fdc, val)     ddi_put32(fdc->c_handlep_aux, \
254                                 ((uint32_t *)fdc->c_auxio_reg), \
255                                     ((uint32_t)(val)))

257 #define Get_auxio(fdc)          ddi_get32(fdc->c_handlep_aux, \
258                                 ((uint32_t *)fdc->c_auxio_reg))

260 /*
261  * DMA registers (sun4u only)
262  */
263 #ifndef _ASM
264 struct  cheerio_dma_reg {
265         uint_t fdc_dcsr;                /* Data Control Status Register */
266         uint_t fdc_dacr;                /* DMA Address Count Registers */
267         uint_t fdc_dbcr;                /* DMA Byte Count Register */
268 };
______unchanged_portion_omitted_
```

```
**********************************************************
   32965 Wed Aug 19 07:25:28 2015
new/usr/src/uts/sun4/io/px/px_ib.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 914 hrtime_t px_ib_msix_retarget_timeout = 120ll * NANOSEC; /* 120 seconds */

 916 /*
 917  * Associate a new CPU with a given MSI/X.
 918  * Operate only on MSI/Xs which are already mapped to devices.
 919  */
 920 int
 921 px_ib_set_msix_target(px_t *px_p, ddi_intr_handle_impl_t *hdlp,
 922     msinum_t msi_num, cpuid_t cpu_id)
 923 {
 924         px_ib_t                 *ib_p = px_p->px_ib_p;
 925         px_msi_state_t          *msi_state_p = &px_p->px_ib_p->ib_msi_state;
 926         dev_info_t              *dip = px_p->px_dip;
 927         dev_info_t              *rdip = hdlp->ih_dip;
 928         msiqid_t                msiq_id, old_msiq_id;
 929         pci_msi_state_t         msi_state;
 930         msiq_rec_type_t         msiq_rec_type;
 931         msi_type_t              msi_type;
 932         px_ino_t                *ino_p;
 933         px_ih_t                 *ih_p, *old_ih_p;
 934         cpuid_t                 old_cpu_id;
 935         hrtime_t                start_time, end_time;
 936         int                     ret = DDI_SUCCESS;
 937         extern const int        _ncpu;
 938         extern cpu_t            *cpu[];

 940         DBG(DBG_IB, dip, "px_ib_set_msix_target: msi_num %x new cpu_id %x\n",
 941             msi_num, cpu_id);

 943         mutex_enter(&cpu_lock);

 945         /* Check for MSI64 support */
 946         if ((hdlp->ih_cap & DDI_INTR_FLAG_MSI64) && msi_state_p->msi_addr64) {
 947                 msiq_rec_type = MSI64_REC;
 948                 msi_type = MSI64_TYPE;
 949         } else {
 950                 msiq_rec_type = MSI32_REC;
 951                 msi_type = MSI32_TYPE;
 952         }

 954         if ((ret = px_lib_msi_getmsiq(dip, msi_num,
 955             &old_msiq_id)) != DDI_SUCCESS) {

 957                 mutex_exit(&cpu_lock);
 958                 return (ret);
 959         }

 961         DBG(DBG_IB, dip, "px_ib_set_msix_target: current msiq 0x%x\n",
 962             old_msiq_id);

 964         if ((ret = px_ib_get_intr_target(px_p,
 965             px_msiqid_to_devino(px_p, old_msiq_id),
 966             &old_cpu_id)) != DDI_SUCCESS) {

 968                 mutex_exit(&cpu_lock);
 969                 return (ret);
 970         }

 972         DBG(DBG_IB, dip, "px_ib_set_msix_target: current cpuid 0x%x\n",
```



```
 973             old_cpu_id);

 975         if (cpu_id == old_cpu_id) {

 977                 mutex_exit(&cpu_lock);
 978                 return (DDI_SUCCESS);
 979         }

 981         /*
 982          * Get lock, validate cpu and write it.
 983          */
 984         if (!((cpu_id < _ncpu) && (cpu[cpu_id] &&
 985             cpu_is_online(cpu[cpu_id])))) {
 986                 /* Invalid cpu */
 987                 DBG(DBG_IB, dip, "px_ib_set_msix_target: Invalid cpuid %x\n",
 988                     cpu_id);

 990                 mutex_exit(&cpu_lock);
 991                 return (DDI_EINVAL);
 992         }

 994         DBG(DBG_IB, dip, "px_ib_set_msix_target: Enabling CPU %d\n", cpu_id);

 996         if ((ret = px_add_msiq_intr(dip, rdip, hdlp,
 997             msiq_rec_type, msi_num, cpu_id, &msiq_id)) != DDI_SUCCESS) {
 998                 DBG(DBG_IB, dip, "px_ib_set_msix_target: Add MSI handler "
 999                     "failed, rdip 0x%p msi 0x%x\n", rdip, msi_num);

1001                 mutex_exit(&cpu_lock);
1002                 return (ret);
1003         }

1005         if ((ret = px_lib_msi_setmsiq(dip, msi_num,
1006             msiq_id, msi_type)) != DDI_SUCCESS) {
1007                 mutex_exit(&cpu_lock);

1009                 (void) px_rem_msiq_intr(dip, rdip,
1010                     hdlp, msiq_rec_type, msi_num, msiq_id);

1012                 return (ret);
1013         }

1015         if ((ret = px_ib_update_intr_state(px_p, rdip, hdlp->ih_inum,
1016             px_msiqid_to_devino(px_p, msiq_id), hdlp->ih_pri,
1017             PX_INTR_STATE_ENABLE, msiq_rec_type, msi_num)) != DDI_SUCCESS) {
1018                 mutex_exit(&cpu_lock);

1020                 (void) px_rem_msiq_intr(dip, rdip,
1021                     hdlp, msiq_rec_type, msi_num, msiq_id);

1023                 return (ret);
1024         }

1026         mutex_exit(&cpu_lock);

1028         /*
1029          * Remove the old handler, but first ensure it is finished.
1030          *
1031          * Each handler sets its PENDING flag before it clears the MSI state.
1032          * Then it clears that flag when finished.  If a re-target occurs while
1033          * the MSI state is DELIVERED, then it is not yet known which of the
1034          * two handlers will take the interrupt.  So the re-target operation
1035          * sets a RETARGET flag on both handlers in that case.  Monitoring both
1036          * flags on both handlers then determines when the old handler can be
1037          * be safely removed.
1038          */
```

```
1039         mutex_enter(&ib_p->ib_ino_lst_mutex);

1041         ino_p = px_ib_locate_ino(ib_p, px_msiqid_to_devino(px_p, old_msiq_id));
1042         old_ih_p = px_ib_intr_locate_ih(px_ib_ino_locate_ipil(ino_p,
1043             hdlp->ih_pri), rdip, hdlp->ih_inum, msiq_rec_type, msi_num);

1045         ino_p = px_ib_locate_ino(ib_p, px_msiqid_to_devino(px_p, msiq_id));
1046         ih_p = px_ib_intr_locate_ih(px_ib_ino_locate_ipil(ino_p, hdlp->ih_pri),
1047             rdip, hdlp->ih_inum, msiq_rec_type, msi_num);

1049         if ((ret = px_lib_msi_getstate(dip, msi_num,
1050             &msi_state)) != DDI_SUCCESS) {
1051                 (void) px_rem_msiq_intr(dip, rdip,
1052                     hdlp, msiq_rec_type, msi_num, msiq_id);

1054                 mutex_exit(&ib_p->ib_ino_lst_mutex);
1055                 return (ret);
1056         }

1058         if (msi_state == PCI_MSI_STATE_DELIVERED) {
1059                 ih_p->ih_intr_flags |= PX_INTR_RETARGET;
1060                 old_ih_p->ih_intr_flags |= PX_INTR_RETARGET;
1061         }

1063         start_time = gethrtime();
1064         while (((ih_p->ih_intr_flags & PX_INTR_RETARGET) &&
1065             (old_ih_p->ih_intr_flags & PX_INTR_RETARGET)) ||
1066             (old_ih_p->ih_intr_flags & PX_INTR_PENDING)) {

1068                 /* Wait for one second */
1069                 delay(drv_sectohz(1));
1069                 delay(drv_usectohz(1000000));

1071                 end_time = gethrtime() - start_time;
1072                 if (end_time > px_ib_msix_retarget_timeout) {
1073                         cmn_err(CE_WARN, "MSIX retarget %x is not completed, "
1074                             "even after waiting %llx ticks\n",
1075                             msi_num, end_time);
1076                         break;
1077                 }
1078         }

1080         ih_p->ih_intr_flags &= ~(PX_INTR_RETARGET);

1082         mutex_exit(&ib_p->ib_ino_lst_mutex);

1084         ret = px_rem_msiq_intr(dip, rdip,
1085             hdlp, msiq_rec_type, msi_num, old_msiq_id);

1087         return (ret);
1088 }
```

_____***unchanged_portion_omitted_***

```
***********************************************************
   15444 Wed Aug 19 07:25:28 2015
new/usr/src/uts/sun4u/cherrystone/os/cherrystone.c
XXXX introduce drv_sectohz
***********************************************************
______unchanged_portion_omitted_

 129 void
 130 load_platform_drivers(void)
 131 {
 132         char            **drv;
 133         dev_info_t      *i2cnexus_dip;
 134         dev_info_t      *keysw_dip = NULL;

 136         static char     *boot_time_drivers[] = {
 137                 "todds1287",
 138                 "mc-us3",
 139                 "ssc050",
 140                 NULL
 141         };

 143         for (drv = boot_time_drivers; *drv; drv++) {
 144                 if (i_ddi_attach_hw_nodes(*drv) != DDI_SUCCESS)
 145                         cmn_err(CE_WARN, "Failed to install \"%s\" driver.",
 146                             *drv);
 147         }

 149         /*
 150          * mc-us3 and ssc050 must stay loaded for plat_get_mem_unum()
 151          * and keyswitch_poll()
 152          */
 153         (void) ddi_hold_driver(ddi_name_to_major("mc-us3"));
 154         (void) ddi_hold_driver(ddi_name_to_major("ssc050"));

 156         /* Gain access into the ssc050_get_port function */
 157         cherry_ssc050_get_port_bit = (int (*) (dev_info_t *, int, int,
 158             uint8_t *, int)) modgetsymvalue("ssc050_get_port_bit", 0);
 159         if (cherry_ssc050_get_port_bit == NULL) {
 160                 cmn_err(CE_WARN, "cannot find ssc050_get_port_bit");
 161                 return;
 162         }

 164         e_ddi_walk_driver("i2c-ssc050", cherry_dev_search, (void *)&keysw_dip);
 165         ASSERT(keysw_dip != NULL);

 167         /*
 168          * prevent detach of i2c-ssc050
 169          */
 170         e_ddi_hold_devi(keysw_dip);

 172         keypoll_timeout_hz = drv_sectohz(10);
 172         keypoll_timeout_hz = drv_usectohz(10 * MICROSEC);
 173         keyswitch_poll(keysw_dip);
 174         abort_seq_handler = cherry_abort_seq_handler;

 176         /*
 177          * Figure out which pcf8584_dip is shared with OBP for the nvram
 178          * device, so the lock can be acquired.
 179          */

 181         i2cnexus_dip = e_ddi_hold_devi_by_path(SHARED_PCF8584_PATH, 0);

 183         ASSERT(i2cnexus_dip != NULL);
 184         shared_pcf8584_dip = ddi_get_parent(i2cnexus_dip);

 186         ndi_hold_devi(shared_pcf8584_dip);
```



```
 187         ndi_rele_devi(i2cnexus_dip);
 188 }
______unchanged_portion_omitted_
```

```
**********************************************************
  210191 Wed Aug 19 07:25:28 2015
new/usr/src/uts/sun4u/cpu/us3_common.c
XXXX introduce drv_sectohz
**********************************************************
_____unchanged_portion_omitted_
2366 #endif  /* CPU_IMP_L1_CACHE_PARITY */

2368 /*
2369  * The cpu_async_log_err() function is called via the [uc]e_drain() function to
2370  * post-process CPU events that are dequeued.  As such, it can be invoked
2371  * from softint context, from AST processing in the trap() flow, or from the
2372  * panic flow.  We decode the CPU-specific data, and take appropriate actions.
2373  * Historically this entry point was used to log the actual cmn_err(9F) text;
2374  * now with FMA it is used to prepare 'flt' to be converted into an ereport.
2375  * With FMA this function now also returns a flag which indicates to the
2376  * caller whether the ereport should be posted (1) or suppressed (0).
2377  */
2378 static int
2379 cpu_async_log_err(void *flt, errorq_elem_t *eqep)
2380 {
2381         ch_async_flt_t *ch_flt = (ch_async_flt_t *)flt;
2382         struct async_flt *aflt = (struct async_flt *)flt;
2383         uint64_t errors;
2384         extern void memscrub_induced_error(void);

2386         switch (ch_flt->flt_type) {
2387         case CPU_INV_AFSR:
2388                 /*
2389                  * If it is a disrupting trap and the AFSR is zero, then
2390                  * the event has probably already been noted. Do not post
2391                  * an ereport.
2392                  */
2393                 if ((aflt->flt_status & ECC_C_TRAP) &&
2394                     (!(aflt->flt_stat & C_AFSR_MASK)))
2395                         return (0);
2396                 else
2397                         return (1);
2398         case CPU_TO:
2399         case CPU_BERR:
2400         case CPU_FATAL:
2401         case CPU_FPUERR:
2402                 return (1);

2404         case CPU_UE_ECACHE_RETIRE:
2405                 cpu_log_err(aflt);
2406                 cpu_page_retire(ch_flt);
2407                 return (1);

2409         /*
2410          * Cases where we may want to suppress logging or perform
2411          * extended diagnostics.
2412          */
2413         case CPU_CE:
2414         case CPU_EMC:
2415                 /*
2416                  * We want to skip logging and further classification
2417                  * only if ALL the following conditions are true:
2418                  *
2419                  *      1. There is only one error
2420                  *      2. That error is a correctable memory error
2421                  *      3. The error is caused by the memory scrubber (in
2422                  *         which case the error will have occurred under
2423                  *         on_trap protection)
2424                  *      4. The error is on a retired page
2425                  *
```



```
2426                  * Note: AFLT_PROT_EC is used places other than the memory
2427                  * scrubber.  However, none of those errors should occur
2428                  * on a retired page.
2429                  */
2430                 if ((ch_flt->afsr_errs &
2431                     (C_AFSR_ALL_ERRS | C_AFSR_EXT_ALL_ERRS)) == C_AFSR_CE &&
2432                     aflt->flt_prot == AFLT_PROT_EC) {

2434                         if (page_retire_check(aflt->flt_addr, NULL) == 0) {
2435                                 if (ch_flt->flt_trapped_ce & CE_CEEN_DEFER) {

2437                                         /*
2438                                          * Since we're skipping logging, we'll need
2439                                          * to schedule the re-enabling of CEEN
2440                                          */
2441                                         (void) timeout(cpu_delayed_check_ce_errors,
2442                                             (void *)(uintptr_t)aflt->flt_inst,
2443                                             drv_sectohz((clock_t)cpu_ceen_delay_secs));
2443                                             drv_usectohz((clock_t)cpu_ceen_delay_secs
2444                                             * MICROSEC));
2444                                 }

2446                                 /*
2447                                  * Inform memscrubber - scrubbing induced
2448                                  * CE on a retired page.
2449                                  */
2450                                 memscrub_induced_error();
2451                                 return (0);
2452                         }
2453                 }

2455                 /*
2456                  * Perform/schedule further classification actions, but
2457                  * only if the page is healthy (we don't want bad
2458                  * pages inducing too much diagnostic activity).  If we could
2459                  * not find a page pointer then we also skip this.  If
2460                  * ce_scrub_xdiag_recirc returns nonzero then it has chosen
2461                  * to copy and recirculate the event (for further diagnostics)
2462                  * and we should not proceed to log it here.
2463                  *
2464                  * This must be the last step here before the cpu_log_err()
2465                  * below - if an event recirculates cpu_ce_log_err() will
2466                  * not call the current function but just proceed directly
2467                  * to cpu_ereport_post after the cpu_log_err() avoided below.
2468                  *
2469                  * Note: Check cpu_impl_async_log_err if changing this
2470                  */
2471                 if (page_retire_check(aflt->flt_addr, &errors) == EINVAL) {
2472                         CE_XDIAG_SETSKIPCODE(aflt->flt_disp,
2473                             CE_XDIAG_SKIP_NOPP);
2474                 } else {
2475                         if (errors != PR_OK) {
2476                                 CE_XDIAG_SETSKIPCODE(aflt->flt_disp,
2477                                     CE_XDIAG_SKIP_PAGEDET);
2478                         } else if (ce_scrub_xdiag_recirc(aflt, ce_queue, eqep,
2479                             offsetof(ch_async_flt_t, cmn_asyncflt))) {
2480                                 return (0);
2481                         }
2482                 }
2483                 /*FALLTHRU*/

2485         /*
2486          * Cases where we just want to report the error and continue.
2487          */
2488         case CPU_CE_ECACHE:
2489         case CPU_UE_ECACHE:
```

```
2490         case CPU_IV:
2491         case CPU_ORPH:
2492                 cpu_log_err(aflt);
2493                 return (1);

2495         /*
2496          * Cases where we want to fall through to handle panicking.
2497          */
2498         case CPU_UE:
2499                 /*
2500                  * We want to skip logging in the same conditions as the
2501                  * CE case.  In addition, we want to make sure we're not
2502                  * panicking.
2503                  */
2504                 if (!panicstr && (ch_flt->afsr_errs &
2505                     (C_AFSR_ALL_ERRS | C_AFSR_EXT_ALL_ERRS)) == C_AFSR_UE &&
2506                     aflt->flt_prot == AFLT_PROT_EC) {
2507                         if (page_retire_check(aflt->flt_addr, NULL) == 0) {
2508                                 /* Zero the address to clear the error */
2509                                 softcall(ecc_page_zero, (void *)aflt->flt_addr);
2510                                 /*
2511                                  * Inform memscrubber - scrubbing induced
2512                                  * UE on a retired page.
2513                                  */
2514                                 memscrub_induced_error();
2515                                 return (0);
2516                         }
2517                 }
2518                 cpu_log_err(aflt);
2519                 break;

2521         default:
2522                 /*
2523                  * If the us3_common.c code doesn't know the flt_type, it may
2524                  * be an implementation-specific code.  Call into the impldep
2525                  * backend to find out what to do: if it tells us to continue,
2526                  * break and handle as if falling through from a UE; if not,
2527                  * the impldep backend has handled the error and we're done.
2528                  */
2529                 switch (cpu_impl_async_log_err(flt, eqep)) {
2530                 case CH_ASYNC_LOG_DONE:
2531                         return (1);
2532                 case CH_ASYNC_LOG_RECIRC:
2533                         return (0);
2534                 case CH_ASYNC_LOG_CONTINUE:
2535                         break; /* continue on to handle UE-like error */
2536                 default:
2537                         cmn_err(CE_WARN, "discarding error 0x%p with "
2538                             "invalid fault type (0x%x)",
2539                             (void *)aflt, ch_flt->flt_type);
2540                         return (0);
2541                 }
2542         }

2544         /* ... fall through from the UE case */

2546         if (aflt->flt_addr != AFLT_INV_ADDR && aflt->flt_in_memory) {
2547                 if (!panicstr) {
2548                         cpu_page_retire(ch_flt);
2549                 } else {
2550                         /*
2551                          * Clear UEs on panic so that we don't
2552                          * get haunted by them during panic or
2553                          * after reboot
2554                          */
2555                         cpu_clearphys(aflt);
```



```
2556                         (void) clear_errors(NULL);
2557                 }
2558         }

2560         return (1);
2561 }
```

_____***unchanged_portion_omitted_***

```
2593 /*
2594  * The cpu_log_err() function is called by cpu_async_log_err() to perform the
2595  * generic event post-processing for correctable and uncorrectable memory,
2596  * E$, and MTag errors.  Historically this entry point was used to log bits of
2597  * common cmn_err(9F) text; now with FMA it is used to prepare 'flt' to be
2598  * converted into an ereport.  In addition, it transmits the error to any
2599  * platform-specific service-processor FRU logging routines, if available.
2600  */
2601 void
2602 cpu_log_err(struct async_flt *aflt)
2603 {
2604         char unum[UNUM_NAMLEN];
2605         int synd_status, synd_code, afar_status;
2606         ch_async_flt_t *ch_flt = (ch_async_flt_t *)aflt;

2608         if (cpu_error_is_ecache_data(aflt->flt_inst, ch_flt->flt_bit))
2609                 aflt->flt_status |= ECC_ECACHE;
2610         else
2611                 aflt->flt_status &= ~ECC_ECACHE;
2612         /*
2613          * Determine syndrome status.
2614          */
2615         synd_status = afsr_to_synd_status(aflt->flt_inst,
2616             ch_flt->afsr_errs, ch_flt->flt_bit);

2618         /*
2619          * Determine afar status.
2620          */
2621         if (pf_is_memory(aflt->flt_addr >> MMU_PAGESHIFT))
2622                 afar_status = afsr_to_afar_status(ch_flt->afsr_errs,
2623                     ch_flt->flt_bit);
2624         else
2625                 afar_status = AFLT_STAT_INVALID;

2627         synd_code = synd_to_synd_code(synd_status,
2628             aflt->flt_synd, ch_flt->flt_bit);

2630         /*
2631          * If afar status is not invalid do a unum lookup.
2632          */
2633         if (afar_status != AFLT_STAT_INVALID) {
2634                 (void) cpu_get_mem_unum_synd(synd_code, aflt, unum);
2635         } else {
2636                 unum[0] = '\0';
2637         }

2639         /*
2640          * Do not send the fruid message (plat_ecc_error_data_t)
2641          * to the SC if it can handle the enhanced error information
2642          * (plat_ecc_error2_data_t) or when the tunable
2643          * ecc_log_fruid_enable is set to 0.
2644          */

2646         if (&plat_ecc_capability_sc_get &&
2647             plat_ecc_capability_sc_get(PLAT_ECC_ERROR_MESSAGE)) {
2648                 if (&plat_log_fruid_error)
2649                         plat_log_fruid_error(synd_code, aflt, unum,
2650                             ch_flt->flt_bit);
```

```
2651         }

2653         if (aflt->flt_func != NULL)
2654                 aflt->flt_func(aflt, unum);

2656         if (afar_status != AFLT_STAT_INVALID)
2657                 cpu_log_diag_info(ch_flt);

2659         /*
2660          * If we have a CEEN error , we do not reenable CEEN until after
2661          * we exit the trap handler. Otherwise, another error may
2662          * occur causing the handler to be entered recursively.
2663          * We set a timeout to trigger in cpu_ceen_delay_secs seconds,
2664          * to try and ensure that the CPU makes progress in the face
2665          * of a CE storm.
2666          */
2667         if (ch_flt->flt_trapped_ce & CE_CEEN_DEFER) {
2668                 (void) timeout(cpu_delayed_check_ce_errors,
2669                     (void *)(uintptr_t)aflt->flt_inst,
2670                     drv_sectohz((clock_t)cpu_ceen_delay_secs));
2671                     drv_usectohz((clock_t)cpu_ceen_delay_secs * MICROSEC));
2671         }
2672 }
______unchanged_portion_omitted_

6065 /*
6066  * Timeout function to reenable CE
6067  */
6068 static void
6069 cpu_delayed_check_ce_errors(void *arg)
6070 {
6071         if (!taskq_dispatch(ch_check_ce_tq, cpu_check_ce_errors, arg,
6072             TQ_NOSLEEP)) {
6073                 (void) timeout(cpu_delayed_check_ce_errors, arg,
6074                     drv_sectohz((clock_t)cpu_ceen_delay_secs));
6075                     drv_usectohz((clock_t)cpu_ceen_delay_secs * MICROSEC));
6075         }
6076 }

6078 /*
6079  * CE Deferred Re-enable after trap.
6080  *
6081  * When the CPU gets a disrupting trap for any of the errors
6082  * controlled by the CEEN bit, CEEN is disabled in the trap handler
6083  * immediately. To eliminate the possibility of multiple CEs causing
6084  * recursive stack overflow in the trap handler, we cannot
6085  * reenable CEEN while still running in the trap handler. Instead,
6086  * after a CE is logged on a CPU, we schedule a timeout function,
6087  * cpu_check_ce_errors(), to trigger after cpu_ceen_delay_secs
6088  * seconds. This function will check whether any further CEs
6089  * have occurred on that CPU, and if none have, will reenable CEEN.
6090  *
6091  * If further CEs have occurred while CEEN is disabled, another
6092  * timeout will be scheduled. This is to ensure that the CPU can
6093  * make progress in the face of CE 'storms', and that it does not
6094  * spend all its time logging CE errors.
6095  */
6096 static void
6097 cpu_check_ce_errors(void *arg)
6098 {
6099         int     cpuid = (int)(uintptr_t)arg;
6100         cpu_t   *cp;

6102         /*
6103          * We acquire cpu_lock.
6104          */
```



```
6105         ASSERT(curthread->t_pil == 0);

6107         /*
6108          * verify that the cpu is still around, DR
6109          * could have got there first ...
6110          */
6111         mutex_enter(&cpu_lock);
6112         cp = cpu_get(cpuid);
6113         if (cp == NULL) {
6114                 mutex_exit(&cpu_lock);
6115                 return;
6116         }
6117         /*
6118          * make sure we don't migrate across CPUs
6119          * while checking our CE status.
6120          */
6121         kpreempt_disable();

6123         /*
6124          * If we are running on the CPU that got the
6125          * CE, we can do the checks directly.
6126          */
6127         if (cp->cpu_id == CPU->cpu_id) {
6128                 mutex_exit(&cpu_lock);
6129                 cpu_check_ce(TIMEOUT_CEEN_CHECK, 0, 0, 0);
6130                 kpreempt_enable();
6131                 return;
6132         }
6133         kpreempt_enable();

6135         /*
6136          * send an x-call to get the CPU that originally
6137          * got the CE to do the necessary checks. If we can't
6138          * send the x-call, reschedule the timeout, otherwise we
6139          * lose CEEN forever on that CPU.
6140          */
6141         if (CPU_XCALL_READY(cp->cpu_id) && (!(cp->cpu_flags & CPU_QUIESCED))) {
6142                 xc_one(cp->cpu_id, (xcfunc_t *)cpu_check_ce,
6143                     TIMEOUT_CEEN_CHECK, 0);
6144                 mutex_exit(&cpu_lock);
6145         } else {
6146                 /*
6147                  * When the CPU is not accepting xcalls, or
6148                  * the processor is offlined, we don't want to
6149                  * incur the extra overhead of trying to schedule the
6150                  * CE timeout indefinitely. However, we don't want to lose
6151                  * CE checking forever.
6152                  *
6153                  * Keep rescheduling the timeout, accepting the additional
6154                  * overhead as the cost of correctness in the case where we get
6155                  * a CE, disable CEEN, offline the CPU during the
6156                  * the timeout interval, and then online it at some
6157                  * point in the future. This is unlikely given the short
6158                  * cpu_ceen_delay_secs.
6159                  */
6160                 mutex_exit(&cpu_lock);
6161                 (void) timeout(cpu_delayed_check_ce_errors,
6162                     (void *)(uintptr_t)cp->cpu_id,
6163                     drv_sectohz((clock_t)cpu_ceen_delay_secs));
6164                     drv_usectohz((clock_t)cpu_ceen_delay_secs * MICROSEC));
6164         }
6165 }
______unchanged_portion_omitted_
```

****
14187 Wed Aug 19 07:25:29 2015
new/usr/src/uts/sun4u/daktari/os/daktari.c
XXXX introduce drv_sectohz
****
_____***unchanged_portion_omitted_***

```
 125 void
 126 load_platform_drivers(void)
 127 {
 128         char **drv;
 129         dev_info_t      *keysw_dip;

 131         static char *boot_time_drivers[] = {
 132                 "hpc3130",
 133                 "todds1287",
 134                 "mc-us3",
 135                 "ssc050",
 136                 "pcisch",
 137                 NULL
 138         };

 140         for (drv = boot_time_drivers; *drv; drv++) {
 141                 if (i_ddi_attach_hw_nodes(*drv) != DDI_SUCCESS)
 142                         cmn_err(CE_WARN, "Failed to install \"%s\" driver.",
 143                             *drv);
 144         }

 146         /*
 147          * mc-us3 & ssc050 must stay loaded for plat_get_mem_unum()
 148          * and keyswitch_poll()
 149          */
 150         (void) ddi_hold_driver(ddi_name_to_major("mc-us3"));
 151         (void) ddi_hold_driver(ddi_name_to_major("ssc050"));

 153         /* Gain access into the ssc050_get_port function */
 154         daktari_ssc050_get_port_bit = (int (*) (dev_info_t *, int, int,
 155             uint8_t *, int)) modgetsymvalue("ssc050_get_port_bit", 0);
 156         if (daktari_ssc050_get_port_bit == NULL) {
 157                 cmn_err(CE_WARN, "cannot find ssc050_get_port_bit");
 158                 return;
 159         }

 161         ddi_walk_devs(ddi_root_node(), daktari_dev_search, (void *)&keysw_dip);
 162         ASSERT(keysw_dip != NULL);

 164         /*
 165          * prevent detach of i2c-ssc050
 166          */
 167         e_ddi_hold_devi(keysw_dip);

 169         keypoll_timeout_hz = drv_sectohz(10);
 169         keypoll_timeout_hz = drv_usectohz(10 * MICROSEC);
 170         keyswitch_poll(keysw_dip);
 171         abort_seq_handler = daktari_abort_seq_handler;
 172 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10401 Wed Aug 19 07:25:29 2015
new/usr/src/uts/sun4u/excalibur/io/xcalwd.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 388 /*
 389  * These are private ioctls for PICL environmental control plug-in
 390  * to use. The plug-in enables the watchdog before performing
 391  * altering fan speeds. It also periodically issues a keepalive
 392  * to the watchdog to cancel and reinstate the watchdog timer.
 393  * The watchdog timeout handler when executed with the watchdog
 394  * enabled sets fans to full blast by calling plat_fan_blast.
 395  */
 396 /*ARGSUSED*/
 397 static int
 398 xcalwd_ioctl(dev_t dev, int cmd, intptr_t arg, int flag,
 399                         cred_t *cred_p, int *rvalp)
 400 {
 401         int             instance;
 402         xcalwd_state_t  *tsp;
 403         int             intvl;
 404         int             o_intvl;
 405         boolean_t       curstate;
 406         timeout_id_t    tid;

 408         if (secpolicy_sys_config(cred_p, B_FALSE) != 0)
 409                 return (EPERM);

 411         instance = getminor(dev);
 412         if (instance < 0)
 413                 return (ENXIO);

 415         tsp = ddi_get_soft_state(xcalwd_statep, instance);
 416         if (tsp == NULL)
 417                 return (ENXIO);

 419         switch (cmd) {
 420         case XCALWD_STOPWATCHDOG:
 421                 /*
 422                  * cancels any pending timer and disables the timer.
 423                  */
 424                 tid = 0;
 425                 mutex_enter(&tsp->lock);
 426                 if (tsp->started == B_FALSE) {
 427                         mutex_exit(&tsp->lock);
 428                         return (0);
 429                 }
 430                 tid = tsp->tid;
 431                 tsp->started = B_FALSE;
 432                 tsp->tid = 0;
 433                 mutex_exit(&tsp->lock);
 434                 if (tid != 0)
 435                         (void) untimeout(tid);
 436                 return (0);
 437         case XCALWD_STARTWATCHDOG:
 438                 if (ddi_copyin((void *)arg, &intvl, sizeof (intvl), flag))
 439                         return (EFAULT);
 440                 if (intvl == 0)
 441                         return (EINVAL);

 443                 mutex_enter(&tsp->lock);
 444                 o_intvl = tsp->intvl;
 445                 mutex_exit(&tsp->lock);
```



```
 447                 if (ddi_copyout((const void *)&o_intvl, (void *)arg,
 448                     sizeof (o_intvl), flag))
 449                         return (EFAULT);

 451                 mutex_enter(&tsp->lock);
 452                 if (tsp->started == B_TRUE) {
 453                         mutex_exit(&tsp->lock);
 454                         return (EINVAL);
 455                 }
 456                 tsp->intvl = intvl;
 457                 tsp->tid = realtime_timeout(xcalwd_timeout,
 458                     (void *)(uintptr_t)instance,
 459                     drv_sectohz(tsp->intvl));
 459                     drv_usectohz(1000000) * tsp->intvl);
 460                 tsp->started = B_TRUE;
 461                 mutex_exit(&tsp->lock);
 462                 return (0);
 463         case XCALWD_KEEPALIVE:
 464                 tid = 0;
 465                 mutex_enter(&tsp->lock);
 466                 tid = tsp->tid;
 467                 tsp->tid = 0;
 468                 mutex_exit(&tsp->lock);
 469                 if (tid != 0)
 470                         (void) untimeout(tid);  /* cancel */

 472                 mutex_enter(&tsp->lock);
 473                 if (tsp->started == B_TRUE)     /* reinstate */
 474                         tsp->tid = realtime_timeout(xcalwd_timeout,
 475                             (void *)(uintptr_t)instance,
 476                             drv_sectohz(tsp->intvl));
 476                             drv_usectohz(1000000) * tsp->intvl);
 477                 mutex_exit(&tsp->lock);
 478                 return (0);
 479         case XCALWD_GETSTATE:
 480                 mutex_enter(&tsp->lock);
 481                 curstate = tsp->started;
 482                 mutex_exit(&tsp->lock);
 483                 if (ddi_copyout((const void *)&curstate, (void *)arg,
 484                     sizeof (curstate), flag))
 485                         return (EFAULT);
 486                 return (0);
 487         default:
 488                 return (EINVAL);
 489         }
 490         /*NOTREACHED*/
 491 }
______unchanged_portion_omitted_
```

```
**********************************************************
   20817 Wed Aug 19 07:25:29 2015
new/usr/src/uts/sun4u/io/pci/pci_ecc.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 560 /*
 561  * Function used to drain pci_ecc_queue, either during panic or after softint
 562  * is generated, to log IO detected ECC errors.
 563  */
 564 /*ARGSUSED*/
 565 void
 566 ecc_err_drain(void *not_used, ecc_errstate_t *ecc_err, errorq_elem_t *eqep)
 567 {
 568         struct async_flt *ecc = &ecc_err->ecc_aflt;
 569         pci_t *pci_p = ecc_err->ecc_p->ecc_pci_cmn_p->pci_p[0];
 570         int ecc_type = ecc_err->ecc_ii_p.ecc_type;

 572         if (pci_p == NULL)
 573                 pci_p = ecc_err->ecc_p->ecc_pci_cmn_p->pci_p[1];

 575         if (ecc->flt_class == RECIRC_BUS_FAULT) {
 576                 /*
 577                  * Perform any additional actions that occur after the
 578                  * ecc_err_cexdiag below and post the ereport.
 579                  */
 580                 ecc->flt_class = BUS_FAULT;
 581                 ecc_err->ecc_err_type = flt_to_error_type(ecc);
 582                 ecc_ereport_post(pci_p->pci_dip, ecc_err);
 583                 return;
 584         }

 586         ecc_cpu_call(ecc, ecc_err->ecc_unum, (ecc_type == CBNINTR_UE) ?
 587             ECC_IO_UE : ECC_IO_CE);

 589         switch (ecc_type) {
 590         case CBNINTR_UE:
 591                 if (ecc_err->ecc_pg_ret == 1) {
 592                         (void) page_retire(ecc->flt_addr, PR_UE);
 593                 }
 594                 ecc_err->ecc_err_type = flt_to_error_type(ecc);
 595                 break;

 597         case CBNINTR_CE:
 598                 /*
 599                  * Setup timeout (if CE detected via interrupt) to
 600                  * re-enable CE interrupts if no more CEs are detected.
 601                  * This is to protect against CE storms.
 602                  */
 603                 if (ecc_ce_delayed &&
 604                     ecc_err->ecc_caller == PCI_ECC_CALL &&
 605                     ecc_err->ecc_p->ecc_to_id == 0) {
 606                         ecc_err->ecc_p->ecc_to_id = timeout(ecc_delayed_ce,
 607                             (void *)ecc_err->ecc_p,
 608                             drv_sectohz((clock_t)ecc_ce_delay_secs);
 608                             drv_usectohz((clock_t)ecc_ce_delay_secs *
 609                             MICROSEC));
 609                 }

 611                 /* ecc_err_cexdiag returns nonzero to recirculate */
 612                 if (CE_XDIAG_EXT_ALG_APPLIED(ecc->flt_disp) &&
 613                     ecc_err_cexdiag(ecc_err, eqep))
 614                         return;
 615                 ecc_err->ecc_err_type = flt_to_error_type(ecc);
 616                 break;
```



```
 617         }

 619         ecc_ereport_post(pci_p->pci_dip, ecc_err);
 620 }
______unchanged_portion_omitted_
```

```
**********************************************************
   95245 Wed Aug 19 07:25:29 2015
new/usr/src/uts/sun4u/io/px/px_hlib.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

3067 static uint_t
3068 oberon_hp_pwron(caddr_t csr_base)
3069 {
3070         volatile uint64_t reg;
3071         boolean_t link_retry, link_up;
3072         int loop, i;

3074         DBG(DBG_HP, NULL, "oberon_hp_pwron the slot\n");

3076         /* Check Leaf Reset status */
3077         reg = CSR_XR(csr_base, ILU_ERROR_LOG_ENABLE);
3078         if (!(reg & (1ull << ILU_ERROR_LOG_ENABLE_SPARE3))) {
3079                 DBG(DBG_HP, NULL, "oberon_hp_pwron fails: leaf not reset\n");
3080                 goto fail;
3081         }

3083         /* Check HP Capable */
3084         if (!CSR_BR(csr_base, TLU_SLOT_CAPABILITIES, HP)) {
3085                 DBG(DBG_HP, NULL, "oberon_hp_pwron fails: leaf not "
3086                     "hotplugable\n");
3087                 goto fail;
3088         }

3090         /* Check Slot status */
3091         reg = CSR_XR(csr_base, TLU_SLOT_STATUS);
3092         if (!(reg & (1ull << TLU_SLOT_STATUS_PSD)) ||
3093             (reg & (1ull << TLU_SLOT_STATUS_MRLS))) {
3094                 DBG(DBG_HP, NULL, "oberon_hp_pwron fails: slot status %lx\n",
3095                     reg);
3096                 goto fail;
3097         }

3099         /* Blink power LED, this is done from pciehpc already */

3101         /* Turn on slot power */
3102         CSR_BS(csr_base, HOTPLUG_CONTROL, PWREN);

3104         /* power fault detection */
3105         delay(drv_usectohz(25000));
3106         CSR_BS(csr_base, TLU_SLOT_STATUS, PWFD);
3107         CSR_BC(csr_base, HOTPLUG_CONTROL, PWREN);

3109         /* wait to check power state */
3110         delay(drv_usectohz(25000));

3112         if (!CSR_BR(csr_base, TLU_SLOT_STATUS, PWFD)) {
3113                 DBG(DBG_HP, NULL, "oberon_hp_pwron fails: power fault\n");
3114                 goto fail1;
3115         }

3117         /* power is good */
3118         CSR_BS(csr_base, HOTPLUG_CONTROL, PWREN);

3120         delay(drv_usectohz(25000));
3121         CSR_BS(csr_base, TLU_SLOT_STATUS, PWFD);
3122         CSR_BS(csr_base, TLU_SLOT_CONTROL, PWFDEN);

3124         /* Turn on slot clock */
3125         CSR_BS(csr_base, HOTPLUG_CONTROL, CLKEN);
```



```
3127         link_up = B_FALSE;
3128         link_retry = B_FALSE;

3130         for (loop = 0; (loop < link_retry_count) && (link_up == B_FALSE);
3131             loop++) {
3132                 if (link_retry == B_TRUE) {
3133                         DBG(DBG_HP, NULL, "oberon_hp_pwron : retry link loop "
3134                             "%d\n", loop);
3135                         CSR_BS(csr_base, TLU_CONTROL, DRN_TR_DIS);
3136                         CSR_XS(csr_base, FLP_PORT_CONTROL, 0x1);
3137                         delay(drv_usectohz(10000));
3138                         CSR_BC(csr_base, TLU_CONTROL, DRN_TR_DIS);
3139                         CSR_BS(csr_base, TLU_DIAGNOSTIC, IFC_DIS);
3140                         CSR_BC(csr_base, HOTPLUG_CONTROL, N_PERST);
3141                         delay(drv_usectohz(50000));
3142                 }

3144                 /* Release PCI-E Reset */
3145                 delay(drv_usectohz(wait_perst));
3146                 CSR_BS(csr_base, HOTPLUG_CONTROL, N_PERST);

3148                 /*
3149                  * Open events' mask
3150                  * This should be done from pciehpc already
3151                  */

3153                 /* Enable PCIE port */
3154                 delay(drv_usectohz(wait_enable_port));
3155                 CSR_BS(csr_base, TLU_CONTROL, DRN_TR_DIS);
3156                 CSR_XS(csr_base, FLP_PORT_CONTROL, 0x20);

3158                 /* wait for the link up */
3159                 /* BEGIN CSTYLED */
3160                 for (i = 0; (i < 2) && (link_up == B_FALSE); i++) {
3161                         delay(drv_usectohz(link_status_check));
3162                         reg = CSR_XR(csr_base, DLU_LINK_LAYER_STATUS);

3164                 if ((((reg >> DLU_LINK_LAYER_STATUS_INIT_FC_SM_STS) &
3165                     DLU_LINK_LAYER_STATUS_INIT_FC_SM_STS_MASK) ==
3166                     DLU_LINK_LAYER_STATUS_INIT_FC_SM_STS_FC_INIT_DONE) &&
3167                     (reg & (1ull << DLU_LINK_LAYER_STATUS_DLUP_STS)) &&
3168                     ((reg &
3169                     DLU_LINK_LAYER_STATUS_LNK_STATE_MACH_STS_MASK) ==
3170                     DLU_LINK_LAYER_STATUS_LNK_STATE_MACH_STS_DL_ACTIVE)) {
3171                         DBG(DBG_HP, NULL, "oberon_hp_pwron : "
3172                             "link is up\n");
3173                         link_up = B_TRUE;
3174                 } else
3175                         link_retry = B_TRUE;

3177                 }
3178                 /* END CSTYLED */
3179         }

3181         if (link_up == B_FALSE) {
3182                 DBG(DBG_HP, NULL, "oberon_hp_pwron fails to enable "
3183                     "PCI-E port\n");
3184                 goto fail2;
3185         }

3187         /* link is up */
3188         CSR_BC(csr_base, TLU_DIAGNOSTIC, IFC_DIS);
3189         CSR_BS(csr_base, FLP_PORT_ACTIVE_STATUS, TRAIN_ERROR);
3190         CSR_BS(csr_base, TLU_UNCORRECTABLE_ERROR_STATUS_CLEAR, TE_P);
3191         CSR_BS(csr_base, TLU_UNCORRECTABLE_ERROR_STATUS_CLEAR, TE_S);
```

```
3192         CSR_BC(csr_base, TLU_CONTROL, DRN_TR_DIS);

3194         /* Restore LUP/LDN */
3195         reg = CSR_XR(csr_base, TLU_OTHER_EVENT_LOG_ENABLE);
3196         if (px_tlu_oe_log_mask & (1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_P))
3197                 reg |= 1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_P;
3198         if (px_tlu_oe_log_mask & (1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_P))
3199                 reg |= 1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_P;
3200         if (px_tlu_oe_log_mask & (1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_S))
3201                 reg |= 1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_S;
3202         if (px_tlu_oe_log_mask & (1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_S))
3203                 reg |= 1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_S;
3204         CSR_XS(csr_base, TLU_OTHER_EVENT_LOG_ENABLE, reg);

3206         /*
3207          * Initialize Leaf
3208          * SPLS = 00b, SPLV = 11001b, i.e. 25W
3209          */
3210         reg = CSR_XR(csr_base, TLU_SLOT_CAPABILITIES);
3211         reg &= ~(TLU_SLOT_CAPABILITIES_SPLS_MASK <<
3212             TLU_SLOT_CAPABILITIES_SPLS);
3213         reg &= ~(TLU_SLOT_CAPABILITIES_SPLV_MASK <<
3214             TLU_SLOT_CAPABILITIES_SPLV);
3215         reg |= (0x19 << TLU_SLOT_CAPABILITIES_SPLV);
3216         CSR_XS(csr_base, TLU_SLOT_CAPABILITIES, reg);

3218         /* Turn on Power LED */
3219         reg = CSR_XR(csr_base, TLU_SLOT_CONTROL);
3220         reg &= ~PCIE_SLOTCTL_PWR_INDICATOR_MASK;
3221         reg = pcie_slotctl_pwr_indicator_set(reg,
3222             PCIE_SLOTCTL_INDICATOR_STATE_ON);
3223         CSR_XS(csr_base, TLU_SLOT_CONTROL, reg);

3225         /* Notify to SCF */
3226         if (CSR_BR(csr_base, HOTPLUG_CONTROL, SLOTPON))
3227                 CSR_BC(csr_base, HOTPLUG_CONTROL, SLOTPON);
3228         else
3229                 CSR_BS(csr_base, HOTPLUG_CONTROL, SLOTPON);

3231         /* Wait for one second */
3232         delay(drv_sectohz(1));
3232         delay(drv_usectohz(1000000));

3234         return (DDI_SUCCESS);

3236 fail2:
3237         /* Link up is failed */
3238         CSR_BS(csr_base, FLP_PORT_CONTROL, PORT_DIS);
3239         CSR_BC(csr_base, HOTPLUG_CONTROL, N_PERST);
3240         delay(drv_usectohz(150));

3242         CSR_BC(csr_base, HOTPLUG_CONTROL, CLKEN);
3243         delay(drv_usectohz(100));

3245 fail1:
3246         CSR_BC(csr_base, TLU_SLOT_CONTROL, PWFDEN);

3248         CSR_BC(csr_base, HOTPLUG_CONTROL, PWREN);

3250         reg = CSR_XR(csr_base, TLU_SLOT_CONTROL);
3251         reg &= ~PCIE_SLOTCTL_PWR_INDICATOR_MASK;
3252         reg = pcie_slotctl_pwr_indicator_set(reg,
3253             PCIE_SLOTCTL_INDICATOR_STATE_OFF);
3254         CSR_XS(csr_base, TLU_SLOT_CONTROL, reg);

3256         CSR_BC(csr_base, TLU_SLOT_STATUS, PWFD);
```



```
3258 fail:
3259         return ((uint_t)DDI_FAILURE);
3260 }

3262 hrtime_t oberon_leaf_reset_timeout = 120ll * NANOSEC;   /* 120 seconds */

3264 static uint_t
3265 oberon_hp_pwroff(caddr_t csr_base)
3266 {
3267         volatile uint64_t reg;
3268         volatile uint64_t reg_tluue, reg_tluce;
3269         hrtime_t start_time, end_time;

3271         DBG(DBG_HP, NULL, "oberon_hp_pwroff the slot\n");

3273         /* Blink power LED, this is done from pciehpc already */

3275         /* Clear Slot Event */
3276         CSR_BS(csr_base, TLU_SLOT_STATUS, PSDC);
3277         CSR_BS(csr_base, TLU_SLOT_STATUS, PWFD);

3279         /* DRN_TR_DIS on */
3280         CSR_BS(csr_base, TLU_CONTROL, DRN_TR_DIS);
3281         delay(drv_usectohz(10000));

3283         /* Disable LUP/LDN */
3284         reg = CSR_XR(csr_base, TLU_OTHER_EVENT_LOG_ENABLE);
3285         reg &= ~((1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_P) |
3286             (1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_P) |
3287             (1ull << TLU_OTHER_EVENT_STATUS_SET_LDN_S) |
3288             (1ull << TLU_OTHER_EVENT_STATUS_SET_LUP_S));
3289         CSR_XS(csr_base, TLU_OTHER_EVENT_LOG_ENABLE, reg);

3291         /* Save the TLU registers */
3292         reg_tluue = CSR_XR(csr_base, TLU_UNCORRECTABLE_ERROR_LOG_ENABLE);
3293         reg_tluce = CSR_XR(csr_base, TLU_CORRECTABLE_ERROR_LOG_ENABLE);
3294         /* All clear */
3295         CSR_XS(csr_base, TLU_UNCORRECTABLE_ERROR_LOG_ENABLE, 0);
3296         CSR_XS(csr_base, TLU_CORRECTABLE_ERROR_LOG_ENABLE, 0);

3298         /* Disable port */
3299         CSR_BS(csr_base, FLP_PORT_CONTROL, PORT_DIS);

3301         /* PCIE reset */
3302         delay(drv_usectohz(10000));
3303         CSR_BC(csr_base, HOTPLUG_CONTROL, N_PERST);

3305         /* PCIE clock stop */
3306         delay(drv_usectohz(150));
3307         CSR_BC(csr_base, HOTPLUG_CONTROL, CLKEN);

3309         /* Turn off slot power */
3310         delay(drv_usectohz(100));
3311         CSR_BC(csr_base, TLU_SLOT_CONTROL, PWFDEN);
3312         CSR_BC(csr_base, HOTPLUG_CONTROL, PWREN);
3313         delay(drv_usectohz(25000));
3314         CSR_BS(csr_base, TLU_SLOT_STATUS, PWFD);

3316         /* write 0 to bit 7 of ILU Error Log Enable Register */
3317         CSR_BC(csr_base, ILU_ERROR_LOG_ENABLE, SPARE3);

3319         /* Set back TLU registers */
3320         CSR_XS(csr_base, TLU_UNCORRECTABLE_ERROR_LOG_ENABLE, reg_tluue);
3321         CSR_XS(csr_base, TLU_CORRECTABLE_ERROR_LOG_ENABLE, reg_tluce);
```

```
3323         /* Power LED off */
3324         reg = CSR_XR(csr_base, TLU_SLOT_CONTROL);
3325         reg &= ~PCIE_SLOTCTL_PWR_INDICATOR_MASK;
3326         reg = pcie_slotctl_pwr_indicator_set(reg,
3327             PCIE_SLOTCTL_INDICATOR_STATE_OFF);
3328         CSR_XS(csr_base, TLU_SLOT_CONTROL, reg);

3330         /* Indicator LED blink */
3331         reg = CSR_XR(csr_base, TLU_SLOT_CONTROL);
3332         reg &= ~PCIE_SLOTCTL_ATTN_INDICATOR_MASK;
3333         reg = pcie_slotctl_attn_indicator_set(reg,
3334             PCIE_SLOTCTL_INDICATOR_STATE_BLINK);
3335         CSR_XS(csr_base, TLU_SLOT_CONTROL, reg);

3337         /* Notify to SCF */
3338         if (CSR_BR(csr_base, HOTPLUG_CONTROL, SLOTPON))
3339                 CSR_BC(csr_base, HOTPLUG_CONTROL, SLOTPON);
3340         else
3341                 CSR_BS(csr_base, HOTPLUG_CONTROL, SLOTPON);

3343         start_time = gethrtime();
3344         /* Check Leaf Reset status */
3345         while (!(CSR_BR(csr_base, ILU_ERROR_LOG_ENABLE, SPARE3))) {
3346                 if ((end_time = (gethrtime() - start_time)) >
3347                     oberon_leaf_reset_timeout) {
3348                         cmn_err(CE_WARN, "Oberon leaf reset is not completed, "
3349                             "even after waiting %llx ticks", end_time);

3351                         break;
3352                 }

3354                 /* Wait for one second */
3355                 delay(drv_sectohz(1));
3355                 delay(drv_usectohz(1000000));
3356         }

3358         /* Indicator LED off */
3359         reg = CSR_XR(csr_base, TLU_SLOT_CONTROL);
3360         reg &= ~PCIE_SLOTCTL_ATTN_INDICATOR_MASK;
3361         reg = pcie_slotctl_attn_indicator_set(reg,
3362             PCIE_SLOTCTL_INDICATOR_STATE_OFF);
3363         CSR_XS(csr_base, TLU_SLOT_CONTROL, reg);

3365         return (DDI_SUCCESS);
3366 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   99351 Wed Aug 19 07:25:29 2015
new/usr/src/uts/sun4u/io/rmclomv.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_


 384 static int
 385 rmclomv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 386 {
 387         int                     instance;
 388         int                     err;
 389         char                    *wdog_state;
 390         int                     attaching = 1;

 392         switch (cmd) {
 393         case DDI_ATTACH:
 394                 /*
 395                  * only allow one instance
 396                  */
 397                 instance = ddi_get_instance(dip);
 398                 if (instance != 0)
 399                         return (DDI_FAILURE);

 401                 err = ddi_create_minor_node(dip, "rmclomv", S_IFCHR,
 402                     instance, DDI_PSEUDO, NULL);
 403                 if (err != DDI_SUCCESS)
 404                         return (DDI_FAILURE);

 406                 /*
 407                  * Register with rmc_comm to prevent it being detached
 408                  * (in the unlikely event that its attach succeeded on a
 409                  * platform whose platmod doesn't lock it down).
 410                  */
 411                 err = rmc_comm_register();
 412                 if (err != DDI_SUCCESS) {
 413                         ddi_remove_minor_node(dip, NULL);
 414                         return (DDI_FAILURE);
 415                 }

 417                 /* Remember the dev info */
 418                 rmclomv_dip = dip;

 420                 /*
 421                  * Add the handlers which watch for unsolicited messages
 422                  * and post event to Sysevent Framework.
 423                  */
 424                 err = rmclomv_add_intr_handlers();
 425                 if (err != DDI_SUCCESS) {
 426                         rmc_comm_unregister();
 427                         ddi_remove_minor_node(dip, NULL);
 428                         rmclomv_dip = NULL;
 429                         return (DDI_FAILURE);
 430                 }

 432                 rmclomv_checkrmc_start();
 433                 rmclomv_refresh_start();

 435                 abort_seq_handler = rmclomv_abort_seq_handler;
 436                 ddi_report_dev(dip);

 438                 /*
 439                  * Check whether we have an application watchdog
 440                  */
 441                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip,
```



```
 442                     DDI_PROP_DONTPASS, RMCLOMV_WATCHDOG_MODE,
 443                     &wdog_state) == DDI_PROP_SUCCESS) {
 444                         if (strcmp(wdog_state, "app") == 0) {
 445                                 rmclomv_watchdog_mode = 1;
 446                                 watchdog_enable = 0;
 447                         }
 448                         else
 449                                 rmclomv_watchdog_mode = 0;
 450                         ddi_prop_free(wdog_state);
 451                 }

 453                 tod_ops.tod_set_watchdog_timer = rmc_set_watchdog_timer;
 454                 tod_ops.tod_clear_watchdog_timer = rmc_clear_watchdog_timer;

 456                 /*
 457                  * Now is a good time to activate hardware watchdog
 458                  * (if one exists).
 459                  */
 460                 mutex_enter(&tod_lock);
 461                 if (watchdog_enable && tod_ops.tod_set_watchdog_timer != NULL)
 462                         err = tod_ops.tod_set_watchdog_timer(0);
 463                 mutex_exit(&tod_lock);
 464                 if (err != 0)
 465                         printf("Hardware watchdog enabled\n");

 467                 /*
 468                  * Set time interval and start timesync routine.
 469                  * Also just this once set the Solaris clock
 470                  * to the RMC clock.
 471                  */
 472                 timesync_interval = drv_sectohz(5 * 60);
 472                 timesync_interval = drv_usectohz(5*60 * MICROSEC);
 473                 plat_timesync((void *) &attaching);

 475                 return (DDI_SUCCESS);
 476         case DDI_RESUME:
 477                 return (DDI_SUCCESS);
 478         default:
 479                 return (DDI_FAILURE);
 480         }
 481 }
______unchanged_portion_omitted_

2935 /* ARGSUSED */
2936 static void
2937 rmclomv_checkrmc(caddr_t arg)
2938 {
2939         callb_cpr_t             cprinfo;
2940         int                     err;
2941         int                     retries;
2942         int                     state;
2943         dp_get_sysinfo_r_t      sysinfo;

2945         CALLB_CPR_INIT(&cprinfo, &rmclomv_checkrmc_lock, callb_generic_cpr,
2946             "rmclomv_checkrmc");

2948         mutex_enter(&rmclomv_checkrmc_lock);
2949         for (;;) {
2950                 /*
2951                  * Initial entry to this for loop is made with
2952                  * rmclomv_checkrmc_sig set to RMCLOMV_PROCESS_NOW. So the
2953                  * following while loop drops through the first time. A
2954                  * timeout call is made just before polling the RMC. Its
2955                  * interrupt routine sustains this loop by injecting additional
2956                  * state changes and cv events.
2957                  */
```

```
2958                /*
2959                 * Wait for someone to tell me to continue.
2960                 */
2961                while (rmclomv_checkrmc_sig == RMCLOMV_CHECKRMC_WAIT) {
2962                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
2963                        cv_wait(&rmclomv_checkrmc_sig_cv,
2964                            &rmclomv_checkrmc_lock);
2965                        CALLB_CPR_SAFE_END(&cprinfo, &rmclomv_checkrmc_lock);
2966                }

2968                mutex_exit(&rmclomv_checkrmc_lock);
2969                /*
2970                 * mustn't hold same lock as timeout called with
2971                 * when cancelling timer
2972                 */
2973                if (timer_id != 0) {
2974                        (void) untimeout(timer_id);
2975                        timer_id = 0;
2976                }
2977                mutex_enter(&rmclomv_checkrmc_lock);

2979                /* RMCLOMV_CHECKRMC_EXITNOW implies signal by _detach(). */
2980                if (rmclomv_checkrmc_sig == RMCLOMV_CHECKRMC_EXITNOW) {
2981                        rmclomv_checkrmc_sig = RMCLOMV_CHECKRMC_WAIT;

2983                        /* rmclomv_checkrmc_lock is held at this point! */
2984                        CALLB_CPR_EXIT(&cprinfo);

2986                        thread_exit();
2987                        /* NOTREACHED */
2988                }

2990                rmclomv_checkrmc_sig = RMCLOMV_CHECKRMC_WAIT;

2992                /*
2993                 * If the RMC is not responding, rmclomv_do_cmd() takes a
2994                 * long time and eventually times out. We conclude that the
2995                 * RMC is broken if it doesn't respond to a number of polls
2996                 * made 60 secs apart. So that the rmclomv_do_cmd() time-out
2997                 * period isn't added to our 60 second timer, make the
2998                 * timeout() call before calling rmclomv_do_cmd().
2999                 */
3000                if (timer_id == 0) {
3001                        timer_id = timeout(rmclomv_checkrmc_wakeup, NULL,
3002                            drv_sectohz(60));
3002                            60 * drv_usectohz(1000000));
3003                }

3005                mutex_exit(&rmclomv_checkrmc_lock);

3007                err = rmclomv_do_cmd(DP_GET_SYSINFO, DP_GET_SYSINFO_R,
3008                    sizeof (sysinfo), NULL, (intptr_t)&sysinfo);
3009                if (err == 0) {
3010                        mutex_enter(&rmclomv_state_lock);
3011                        state = rmclomv_rmc_state;
3012                        /* successful poll, reset fail count */
3013                        rmclomv_rmcfailcount = 0;
3014                        mutex_exit(&rmclomv_state_lock);

3016                        if (state != RMCLOMV_RMCSTATE_OK) {
3017                                rmclomv_refresh_wakeup();
3018                        }
3019                }
3020                if ((err != 0) &&
3021                    (rmclomv_rmc_error != RMCLOMV_RMCSTATE_DOWNLOAD)) {
3022                        /*
```



```
3023                         * Failed response or no response from RMC.
3024                         * Count the failure.
3025                         * If threshold exceeded, send a DR event.
3026                         */
3027                        mutex_enter(&rmclomv_state_lock);
3028                        retries = rmclomv_rmcfailcount;
3029                        state = rmclomv_rmc_state;
3030                        if (retries == RMCLOMV_RMCFAILTHRESHOLD)
3031                                rmclomv_rmc_state = RMCLOMV_RMCSTATE_FAILED;
3032                        if (rmclomv_rmcfailcount <= RMCLOMV_RMCFAILTHRESHOLD)
3033                                rmclomv_rmcfailcount++;
3034                        mutex_exit(&rmclomv_state_lock);

3036                        if (retries == RMCLOMV_RMCFAILTHRESHOLD) {
3037                                cmn_err(CE_WARN, "SC %s responding",
3038                                    state == RMCLOMV_RMCSTATE_OK ?
3039                                    "has stopped" : "is not");
3040                                refresh_name_cache(B_TRUE);
3041                                rmclomv_dr_data_handler(str_sc, SE_NO_HINT);
3042                        }
3043                }

3045                /*
3046                 * Re-enter the lock to prepare for another iteration.
3047                 * We must have the lock here to protect rmclomv_checkrmc_sig.
3048                 */
3049                mutex_enter(&rmclomv_checkrmc_lock);
3050        }
3051 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15297 Wed Aug 19 07:25:30 2015
new/usr/src/uts/sun4u/littleneck/io/pcf8574_lneck.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 497 static int
 498 pcf8574_do_attach(dev_info_t *dip)
 499 {
 500         struct pcf8574_unit *unitp;
 501         int instance, err;
 502         uint_t len;
 503         int *regs;

 505         instance = ddi_get_instance(dip);

 507         if (ddi_soft_state_zalloc(pcf8574soft_statep, instance) != 0) {
 508                 cmn_err(CE_WARN, "%s%d: failed to zalloc softstate\n",
 509                     ddi_get_name(dip), instance);
 510                 return (DDI_FAILURE);
 511         }

 513         unitp = ddi_get_soft_state(pcf8574soft_statep, instance);

 515         if (unitp == NULL) {
 516                 cmn_err(CE_WARN, "%s%d: unitp not filled\n",
 517                     ddi_get_name(dip), instance);
 518                 return (ENOMEM);
 519         }

 521         (void) snprintf(unitp->pcf8574_name, sizeof (unitp->pcf8574_name),
 522             "%s%d", ddi_node_name(dip), instance);


 525         if (ddi_create_minor_node(dip, "pcf8574", S_IFCHR, instance,
 526             "ddi_i2c:ioexp", NULL) == DDI_FAILURE) {
 527                 cmn_err(CE_WARN, "%s ddi_create_minor_node failed for "
 528                     "%s\n", unitp->pcf8574_name, "pcf8574");
 529                 ddi_soft_state_free(pcf8574soft_statep, instance);

 531                 return (DDI_FAILURE);
 532         }

 534         if (i2c_client_register(dip, &unitp->pcf8574_hdl) != I2C_SUCCESS) {
 535                 ddi_remove_minor_node(dip, NULL);
 536                 ddi_soft_state_free(pcf8574soft_statep, instance);

 538                 return (DDI_FAILURE);
 539         }

 541         err = ddi_prop_lookup_int_array(DDI_DEV_T_ANY, dip,
 542             DDI_PROP_DONTPASS,
 543             "reg", (int **)&regs, &len);
 544         if (err != DDI_PROP_SUCCESS) {
 545                 return (DDI_FAILURE);
 546         }

 548         /*
 549          * regs[0] contains the bus number and regs[1] contains the device
 550          * address of the i2c device. 0x7c is the device address of the
 551          * i2c device from which the key switch position is read.
 552          */
 553         if (regs[0] == 0 && regs[1] == 0x7c) {
 554                 abort_seq_handler = littleneck_abort_seq_handler;
 555                 keypoll_timeout_hz =
```



```
 556                     drv_sectohz(LNECK_KEY_POLL_INTVL);
 556                     drv_usectohz(LNECK_KEY_POLL_INTVL * MICROSEC);
 557                 littleneck_ks_poll(unitp);
 558         }

 560         ddi_prop_free(regs);

 562         mutex_init(&unitp->pcf8574_mutex, NULL, MUTEX_DRIVER, NULL);

 564         return (DDI_SUCCESS);
 565 }
______unchanged_portion_omitted_
```

```
**********************************************************
  207126 Wed Aug 19 07:25:30 2015
new/usr/src/uts/sun4u/montecarlo/io/scsb.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

2664 /*
2665  * called with mutex held from
2666  * scsb_attach()        with int_fru_ptr == NULL
2667  * scsb_intr()          with int_fru_ptr == info for FRU that caused interrupt
2668  */
2669 static int
2670 scsb_set_scfg_pres_leds(scsb_state_t *scsb, fru_info_t *int_fru_ptr)
2671 {
2672         int             i, error = 0;
2673         int             cfg_idx, led_idx, blink_idx, lid, bid;
2674         int             cfg_bit, led_bit;
2675         uchar_t         *puc, reg, led_reg, led_data[SCSB_LEDDATA_REGISTERS];
2676         uchar_t         blink_bit, blink_reg, blink[SCSB_LEDDATA_REGISTERS];
2677         uchar_t         update_reg = 0;
2678         scsb_utype_t    fru_type;
2679         fru_info_t      *fru_ptr;

2681         if (scsb->scsb_state & SCSB_FROZEN &&
2682             !(scsb->scsb_state & SCSB_IN_INTR)) {
2683                 return (EAGAIN);
2684         }
2685         for (i = 0; i < SCTRL_LED_OK_NUMREGS; ++i) {
2686                 led_data[i] = 0;
2687                 blink[i] = 0;
2688         }
2689         led_reg = SCSB_REG_ADDR(SCTRL_LED_OK_BASE);
2690         reg = SCSB_REG_ADDR(SCTRL_BLINK_OK_BASE);
2691         lid = SCSB_REG_INDEX(led_reg);          /* the LED Index Delta */
2692         bid = SCSB_REG_INDEX(reg);              /* the Blink Index Delta */
2693         blink_reg = 0;
2694         if (int_fru_ptr != NULL) {
2695                 update_reg = int_fru_ptr->i2c_info->ledata_reg;
2696         }
2697         for (fru_type = 0; fru_type < SCSB_UNIT_TYPES; ++fru_type) {
2698                 int     is_present;
2699                 fru_ptr = mct_system_info.fru_info_list[fru_type];
2700                 for (; fru_ptr != NULL; fru_ptr = fru_ptr->next) {
2701                         is_present = 0;
2702                         if (fru_type == SLOT && (scsb->scsb_state &
2703                             SCSB_APP_SLOTLED_CTRL))
2704                                 break;
2705                         if (fru_ptr->i2c_info == NULL)
2706                                 continue;
2707                         if ((led_reg = fru_ptr->i2c_info->ledata_reg) == 0) {
2708                                 /*
2709                                  * No LED exceptions: SSB,CRTM,PRTM
2710                                  */
2711                                 continue;
2712                         }
2713                         if (update_reg && update_reg != led_reg)
2714                                 continue;
2715                         led_idx = SCSB_REG_INDEX(led_reg) - lid;
2716                         led_bit = fru_ptr->i2c_info->ledata_bit;
2717                         if ((reg = fru_ptr->i2c_info->syscfg_reg) == 0) {
2718                                 if (fru_type != SCB)
2719                                         continue;
2720                                 /*
2721                                  * exception: SCB
2722                                  */
```



```
2723                                 if (scsb->scsb_state & SCSB_SCB_PRESENT) {
2724                                         led_data[led_idx] |= 1 << led_bit;
2725                                         is_present = 1;
2726                                 } else {
2727                                         led_data[led_idx] &= ~(1 << led_bit);
2728                                 }
2729                                 if (IS_SCB_P10)
2730                                         continue;
2731                         } else {
2732                                 cfg_idx = SCSB_REG_INDEX(reg);
2733                                 cfg_bit = fru_ptr->i2c_info->syscfg_bit;
2734                                 if (scsb->scsb_data_reg[cfg_idx] &
2735                                     (1 << cfg_bit)) {
2736                                         is_present = 1;
2737                                 }
2738                         }
2739                         if (is_present) {
2740                                 /*
2741                                  * If the FRU is a Power Supply, AND
2742                                  * the call is from scsb_attach() OR
2743                                  * from scsb_intr() and FRUs match,
2744                                  * turn it on.
2745                                  */
2746                                 if (fru_type == PS && (int_fru_ptr == NULL ||
2747                                     (int_fru_ptr == fru_ptr))) {
2748                                         pao.scsb = scsb;
2749                                         pao.utype = fru_type;
2750                                         pao.unit = fru_ptr->fru_unit;
2751 #ifdef  PS_ON_DELAY
2752                                         /*
2753                                          * HW recommended not implementing
2754                                          * this delay for now.
2755                                          * The code is tested on PSUs:
2756                                          *      -06
2757                                          *      -07 rev 2
2758                                          *      -08 plus
2759                                          */
2760                                         if (int_fru_ptr) {
2761                                                 /*
2762                                                  * Hot insertion, so give it
2763                                                  * the 3 seconds it needs to
2764                                                  * become stable
2765                                                  */
2766                                                 if (!scsb->scsb_btid)
2767                                                         scsb->scsb_btid =
2768                                                             timeout(
2769                                                             scsb_ps_auto_on,
2770                                                             &pao, drv_sectohz(4)
2770                                                             &pao, (4 *
2771                                                             drv_usectohz(
2772                                                             1000000)));
2771                                         } else
2772 #endif  /* PS_ON_DELAY */
2773                                                 scsb_ps_auto_on((void *)&pao);
2774                                 }
2775                                 /*
2776                                  * Special SLOT handling.
2777                                  * Make sure the OK LED is on for the CPU Slot
2778                                  * and for the FTC (CFTM) Slot for MonteCarlo.
2779                                  * Both will report as FRU_PRESENT.
2780                                  */
2781                                 if (fru_type != SLOT || (fru_type == SLOT &&
2782                                     (fru_ptr->fru_type ==
2783                                     (scsb_utype_t)OC_CPU ||
2784                                     fru_ptr->fru_type ==
2785                                     (scsb_utype_t)OC_CTC))) {
```

```
2786                                    /*
2787                                     * Set OK (green) LED register bit
2788                                     */
2789                                    led_data[led_idx] |= 1 << led_bit;
2790                            }
2791                            if (IS_SCB_P10)
2792                                    continue;
2793                            /*
2794                             * Turn off BLINK register bit.
2795                             * If single register update, then save the
2796                             * corresponding blink register in blink_reg.
2797                             */
2798                            reg = fru_ptr->i2c_info->blink_reg;
2799                            if (!reg)
2800                                    continue;
2801                            blink_bit = fru_ptr->i2c_info->blink_bit;
2802                            blink_idx = SCSB_REG_INDEX(reg) - bid;
2803                            blink[blink_idx] |= 1 << blink_bit;
2804                            if (update_reg && update_reg == led_reg)
2805                                    blink_reg = reg;
2806                    }
2807            }
2808    }
2809    if (update_reg) {
2810            reg = update_reg;
2811            i = SCSB_REG_INDEX(reg);
2812            puc = &led_data[i - lid];
2813            i = 1;
2814    } else {
2815            reg = SCSB_REG_ADDR(SCTRL_LED_OK_BASE);
2816            puc = led_data;
2817            i = SCTRL_LED_OK_NUMREGS;
2818    }
2819    if (scsb_debug & 0x0100) {
2820            cmn_err(CE_NOTE, "scsb_set_scfg_pres(): writing %d bytes "
2821                "to 0x%x", i, reg);
2822    }
2823    if ((error = scsb_rdwr_register(scsb, I2C_WR, reg, i, puc, 1)) != 0) {
2824            if (scsb_debug & 0x0102)
2825                    cmn_err(CE_NOTE, "scsb_set_scfg_pres(): "
2826                        "I2C write to 0x%x failed", reg);
2827            error = EIO;
2828    } else {
2829            /*
2830             * Now see which BLINK bits need to be turned off for the
2831             * corresponding OK LED bits.
2832             */
2833            reg = SCSB_REG_ADDR(SCTRL_BLINK_OK_BASE);
2834            for (i = 0; i < SCTRL_BLINK_NUMREGS; ++i, ++reg) {
2835                    if (blink_reg && blink_reg != reg)
2836                            continue;
2837                    if (!blink[i]) {
2838                            continue;
2839                    }
2840                    if (scsb_debug & 0x0100) {
2841                            cmn_err(CE_NOTE, "scsb_set_scfg_pres(): turn "
2842                                "OFF Blink bits 0x%x in 0x%x",
2843                                blink[i], reg);
2844                    }
2845                    if (scsb_write_mask(scsb, reg, 0, 0, blink[i])) {
2846                            if (scsb_debug & 0x0102)
2847                                    cmn_err(CE_NOTE,
2848                                        "scsb_set_scfg_pres(): "
2849                                        "Write to 0x%x failed", reg);
2850                            error = EIO;
2851                            break;
```



```
2852                    }
2853            }
2854    }
2855    return (error);
2856 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   72873 Wed Aug 19 07:25:30 2015
new/usr/src/uts/sun4u/serengeti/io/sgsbbc_mailbox.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 838 /*
 839  * Client Interface
 840  *
 841  * The main interface will be
 842  *
 843  * sbbc_mbox_request_response(sbbc_msg_t *request,
 844  *                       sbbc_msg_t *response, time_t wait_time)
 845  *
 846  * 1) the client calls request_response
 847  * 2) a new unique msg ID is assigned for that msg
 848  * 3) if there is space available in the outbox
 849  *    - the request msg is written to the mbox_out mailbox
 850  *      and the mailbox info updated.
 851  *    - allocate a sbbc_msg_waiter struct for this
 852  *      message, initialise the w_cv condvar.
 853  *    - get the mailbox mbox_wait_lock mutex for this
 854  *      message type
 855  *    - the response msg is put on the mbox_wait_list for
 856  *      that message type to await the SC's response
 857  *    - wait on the w_cv condvar protected by the
 858  *      mbox_wait_lock
 859  *    - SBBC_MAILBOX_OUT interrupt is sent to the SC
 860  *
 861  * 4) if no space in the outbox,
 862  *    - the request message blocks waiting
 863  *      for a SBBC_MAILBOX_SPACE_OUT interrupt
 864  *      It will block on the mailbox mb_full condvar.
 865  *    - go to (3) above
 866  * 5) When we get a SBBC_MAILBOX_IN interrupt.
 867  *    - read the message ID of the next message (FIFO)
 868  *    - find that ID on the wait list
 869  *    - no wait list entry => unsolicited message. If theres
 870  *      a handler, trigger it
 871  *    - if someone is waiting, read the message in from
 872  *      SRAM, handling fragmentation, wraparound, etc
 873  *    - if the whole message has been read, signal
 874  *      the waiter
 875  *    - read next message until mailbox empty
 876  *    - send SBBC_MAILBOX_SPACE_IN interrupt to the SC
 877  *
 878  * 6) If a response is required and none is received, the client
 879  *      will timeout after <wait_time> seconds and the message
 880  *      status will be set to ETIMEDOUT.
 881  */
 882 int
 883 sbbc_mbox_request_response(sbbc_msg_t *request,
 884                 sbbc_msg_t *response, time_t wait_time)
 885 {

 887         struct sbbc_msg_waiter  *waiter;
 888         uint_t                  msg_id;
 889         int                     rc = 0;
 890         int                     flags;
 891         uint16_t                msg_type;
 892         clock_t                 stop_time;
 893         clock_t                 clockleft;
 894         kmutex_t                *mbox_wait_lock;
 895         kmutex_t                *mb_lock;
 896         static fn_t             f = "sbbc_mbox_request_response";
```



```
 898         if ((request == NULL) ||
 899                 (request->msg_type.type >= SBBC_MBOX_MSG_TYPES) ||
 900                 ((response != NULL) &&
 901                 (response->msg_type.type >= SBBC_MBOX_MSG_TYPES)))
 902                 return (EINVAL);

 904         msg_type = request->msg_type.type;

 906         /*
 907          * Verify that we have already set up the master sbbc
 908          */
 909         if (master_mbox == NULL)
 910                 return (ENXIO);
 911         mbox_wait_lock = &master_mbox->mbox_wait_lock[msg_type];

 913         flags = WAIT_FOR_REPLY|WAIT_FOR_SPACE;

 915         /*
 916          * We want to place a lower limit on the shortest amount of time we
 917          * will wait before timing out while communicating with the SC via
 918          * the mailbox.
 919          */
 920         if (wait_time < sbbc_mbox_min_timeout)
 921                 wait_time = sbbc_mbox_default_timeout;

 923         stop_time = ddi_get_lbolt() + drv_sectohz(wait_time);
 923         stop_time = ddi_get_lbolt() + wait_time * drv_usectohz(MICROSEC);

 925         /*
 926          * If there is a message being processed, sleep until it is our turn.
 927          */
 928         mutex_enter(&outbox_queue_lock);

 930         /*
 931          * allocate an ID for this message, let it wrap
 932          * around transparently.
 933          * msg_id == 0 is unsolicited message
 934          */
 935         msg_id = ++(master_mbox->mbox_msg_id);
 936         if (msg_id == 0)
 937                 msg_id = ++(master_mbox->mbox_msg_id);

 939         SGSBBC_DBG_MBOX("%s: msg_id = 0x%x, msg_len = 0x%x\n",
 940                 f, msg_id, request->msg_len);

 942         /*
 943          * A new message can actually grab the lock before the thread
 944          * that has just been signaled.  Therefore, we need to double
 945          * check to make sure that outbox_busy is not already set
 946          * after we wake up.
 947          *
 948          * Potentially this could mean starvation for certain unfortunate
 949          * threads that keep getting woken up and putting back to sleep.
 950          * But the window of such contention is very small to begin with.
 951          */
 952         while (outbox_busy) {

 954                 clockleft = cv_timedwait(&outbox_queue, &outbox_queue_lock,
 955                         stop_time);

 957                 SGSBBC_DBG_MBOX("%s: msg_id = 0x%x is woken up\n", f, msg_id);

 959                 /*
 960                  * If we have timed out, set status to ETIMEOUT and return.
 961                  */
```

```
962                 if (clockleft < 0) {
963                         SGSBBC_DBG_MBOX("%s: msg_id = 0x%x has timed out\n",
964                             f, msg_id);
965                         cmn_err(CE_NOTE,
966                                 "Timed out obtaining SBBC outbox lock");
967                         request->msg_status = ETIMEDOUT;
968                         if (response != NULL)
969                                 response->msg_status = ETIMEDOUT;
970                         mutex_exit(&outbox_queue_lock);
971                         return (ETIMEDOUT);
972                 }
973         }

975         outbox_busy = 1;
976         mutex_exit(&outbox_queue_lock);

978         /*
979          * We are only locking the OutBox from here, not the whole
980          * mailbox. This is based on the assumption of
981          * complete separation of mailboxes - outbox is
982          * read/write, inbox is read-only.
983          * We only ever update the producer for the
984          * outbox and the consumer for the inbox.
985          */
986         mb_lock = &master_mbox->mbox_out->mb_lock;
987         mutex_enter(mb_lock);

989         /*
990          * No response expected ? Just send the message and return
991          */
992         if (response == NULL) {
993                 rc = sbbc_mbox_send_msg(request, flags, msg_id, wait_time,
994                         stop_time);
995                 SGSBBC_DBG_MBOX("%s: msg_id = 0x%x send rc = %d\n",
996                     f, msg_id, rc);

998                 wakeup_next();

1000                mutex_exit(mb_lock);
1001                request->msg_status = rc;
1002                return (rc);
1003        }

1005        /*
1006         * allocate/initialise a waiter
1007         */
1008        waiter = kmem_zalloc(sizeof (struct sbbc_msg_waiter), KM_NOSLEEP);

1010        if (waiter == (struct sbbc_msg_waiter *)NULL) {
1011                cmn_err(CE_WARN, "SBBC Mailbox can't allocate waiter\n");

1013                wakeup_next();

1015                mutex_exit(mb_lock);
1016                return (ENOMEM);
1017        }

1019        waiter->w_id = 0;       /* Until we get an ID from the send */
1020        waiter->w_msg = response;
1021        waiter->w_msg->msg_status = EINPROGRESS;

1023        cv_init(&waiter->w_cv, NULL, CV_DEFAULT, NULL);

1025        rc = sbbc_mbox_send_msg(request, flags, msg_id, wait_time, stop_time);

1027        wakeup_next();
```



```
1029        if (rc != 0) {

1031                request->msg_status = response->msg_status = rc;
1032                mutex_exit(mb_lock);

1034                /* Free the waiter */
1035                cv_destroy(&waiter->w_cv);
1036                kmem_free(waiter, sizeof (struct sbbc_msg_waiter));

1038                SGSBBC_DBG_MBOX("%s: msg_id = 0x%x send rc = %d\n",
1039                    f, msg_id, rc);

1041                return (rc);
1042        }

1044        waiter->w_id = msg_id;

1046        /*
1047         * Lock this waiter list and add the waiter
1048         */
1049        mutex_enter(mbox_wait_lock);

1051        if (master_mbox->mbox_wait_list[msg_type] == NULL) {
1052                master_mbox->mbox_wait_list[msg_type] = waiter;
1053                waiter->w_next = NULL;
1054        } else {
1055                struct sbbc_msg_waiter  *tmp;
1056                tmp = master_mbox->mbox_wait_list[msg_type];
1057                master_mbox->mbox_wait_list[msg_type] = waiter;
1058                waiter->w_next = tmp;
1059        }

1061        mutex_exit(mb_lock);

1063        /*
1064         * wait here for a response to our message
1065         * holding the mbox_wait_lock for the list ensures
1066         * that the interrupt handler can't get in before
1067         * we block.
1068         * NOTE: We use the request msg_type for the
1069         *       the wait_list. This ensures that  the
1070         *       msg_type won't change.
1071         */
1072        clockleft = cv_timedwait(&waiter->w_cv, mbox_wait_lock, stop_time);

1074        SGSBBC_DBG_MBOX("%s: msg_id = 0x%x is woken up for response\n",
1075                f, msg_id);

1077        /*
1078         * If we have timed out, set msg_status to ETIMEDOUT,
1079         * and remove the waiter from the waiter list.
1080         */
1081        if (clockleft < 0) {
1082                /*
1083                 * Remove the waiter from the waiter list.
1084                 * If we can't find the waiter in the list,
1085                 * 1. msg_status == EINPROGRESS
1086                 *    It is being processed.  We will give it
1087                 *    a chance to finish.
1088                 * 2. msg_status != EINPROGRESS
1089                 *    It is done processing.  We can safely
1090                 *    remove it.
1091                 * If we can find the waiter, it has timed out.
1092                 */
1093                SGSBBC_DBG_MBOX("%s: msg_id = 0x%x has timed out\n",
```

```
1094                        f, msg_id);
1095                if (mbox_find_waiter(msg_type, msg_id) == NULL) {
1096                        if (waiter->w_msg->msg_status == EINPROGRESS) {
1097                                SGSBBC_DBG_MBOX("%s: Waiting for msg_id = 0x%x "
1098                                        "complete.\n", f, msg_id);
1099                                cv_wait(&waiter->w_cv, mbox_wait_lock);
1100                        }
1101                } else {
1102                        SGSBBC_DBG_MBOX("%s: setting msg_id = 0x%x "
1103                                "to ETIMEDOUT\n", f, msg_id);
1104                        cmn_err(CE_NOTE, "Timed out waiting for SC response");
1105                        rc = waiter->w_msg->msg_status = ETIMEDOUT;
1106                }
1107        }

1109        /*
1110         * lose the waiter
1111         */
1112        cv_destroy(&waiter->w_cv);
1113        kmem_free(waiter, sizeof (struct sbbc_msg_waiter));

1115        mutex_exit(mbox_wait_lock);

1117        return (rc);

1119 }
```

_____***unchanged_portion_omitted_***

****
27155 Wed Aug 19 07:25:30 2015
new/usr/src/uts/sun4u/starfire/cvc/cvc.c
XXXX introduce drv_sectohz
****
_____unchanged_portion_omitted_

```
 962 /*
 963  * cvc_getstr()
 964  *      Poll BBSRAM for console input while available.
 965  */
 966 static void
 967 cvc_getstr(char *cp)
 968 {
 969         short           count;
 970         volatile char   *lp;

 972         mutex_enter(&cvc_bbsram_input_mutex);
 973         /* Poll BBSRAM for input */
 974         do {
 975                 if (stop_bbsram) {
 976                         *cp = '\0';     /* set string to zero-length */
 977                         mutex_exit(&cvc_bbsram_input_mutex);
 978                         return;
 979                 }
 980                 /*
 981                  * Use a smaller delay between checks of BBSRAM for input
 982                  * when cvcd/cvcredir are not running or "via_bbsram" has
 983                  * been set.
 984                  * We don't go away completely when i/o is going through the
 985                  * network via cvcd since a command may be sent via BBSRAM
 986                  * to switch if the network is down or hung.
 987                  */
 988                 if ((cvcoutput_q == NULL) || (via_bbsram))
 989                         delay(drv_usectohz(100000));
 990                 else
 991                         delay(drv_sectohz(1));
 991                         delay(drv_usectohz(1000000));
 992                 cvc_bbsram_ops(BBSRAM_CONTROL_REG);
 993                 count = BBSRAM_INPUT_COUNT;
 994         } while (count == 0);

 996         lp = BBSRAM_INPUT_BUF - count;

 998         while (count--) {
 999                 *cp++ = *lp++;
1000         }
1001         *cp = '\0';

1003         BBSRAM_INPUT_COUNT = 0;
1004         mutex_exit(&cvc_bbsram_input_mutex);
1005 }
```

_____unchanged_portion_omitted_

```
***********************************************************
   42273 Wed Aug 19 07:25:30 2015
new/usr/src/uts/sun4u/sunfire/io/ac.c
XXXX introduce drv_sectohz
***********************************************************
______unchanged_portion_omitted_

1497 static int
1498 ac_reset_timeout(int rw)
1499 {
1500         mutex_enter(&ac_hot_plug_mode_mutex);

1502         if ((ac_hot_plug_timeout == (timeout_id_t)NULL) &&
1503             (rw == KSTAT_READ)) {
1504                 /*
1505                  * We are in hot-plug mode. A kstat_read is not
1506                  * going to affect this. return 0 to allow the
1507                  * kstat_read to continue.
1508                  */
1509                 mutex_exit(&ac_hot_plug_mode_mutex);
1510                 return (0);

1512         } else if ((ac_hot_plug_timeout == (timeout_id_t)NULL) &&
1513             (rw == KSTAT_WRITE)) {
1514                 /*
1515                  * There are no pending timeouts and we have received a
1516                  * kstat_write request so we must be transitioning
1517                  * from "hot-plug" mode to non "hot-plug" mode.
1518                  * Try to lock all boards before allowing the kstat_write.
1519                  */
1520                 if (ac_enter_transition() == TRUE)
1521                         fhc_bdlist_unlock();
1522                 else {
1523                         /* cannot lock boards so fail */
1524                         mutex_exit(&ac_hot_plug_mode_mutex);
1525                         return (-1);
1526                 }

1528                 /*
1529                  * We need to display a Warning about hot-plugging any
1530                  * boards. This message is only needed when we are
1531                  * transitioning out of "hot-plug" mode.
1532                  */
1533                 cmn_err(CE_WARN, "This machine is being taken out of "
1534                     "hot-plug mode.");
1535                 cmn_err(CE_CONT, "Do not attempt to hot-plug boards "
1536                     "or power supplies in this system until further notice.");

1538         } else if (ac_hot_plug_timeout != (timeout_id_t)NULL) {
1539                 /*
1540                  * There is a pending timeout so we must already be
1541                  * in non "hot-plug" mode. It doesn't matter if the
1542                  * kstat request is a read or a write.
1543                  *
1544                  * We need to cancel the existing timeout.
1545                  */
1546                 (void) untimeout(ac_hot_plug_timeout);
1547                 ac_hot_plug_timeout = NULL;
1548         }

1550         /*
1551          * create a new timeout.
1552          */
1553         ac_hot_plug_timeout = timeout(ac_timeout, NULL,
1554             drv_sectohz(ac_hot_plug_timeout_interval));
1554             drv_usectohz(ac_hot_plug_timeout_interval * 1000000));
```



```
1556         mutex_exit(&ac_hot_plug_mode_mutex);
1557         return (0);
1558 }
______unchanged_portion_omitted_
```

```
**********************************************************
   84103 Wed Aug 19 07:25:31 2015
new/usr/src/uts/sun4u/sunfire/io/sysctrl.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 391 static int
 392 sysctrl_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
 393 {
 394         struct sysctrl_soft_state *softsp;
 395         int instance;
 396         uchar_t tmp_reg;
 397         dev_info_t *dip;
 398         char *propval;
 399         int proplen;
 400         int slot_num;
 401         int start;              /* start index for scan loop */
 402         int limit;              /* board number limit for scan loop */
 403         int incr;               /* amount to incr each pass thru loop */
 404         void set_clockbrd_info(void);


 407         switch (cmd) {
 408         case DDI_ATTACH:
 409                 break;

 411         case DDI_RESUME:
 412                 /* XXX see sysctrl:DDI_SUSPEND for special h/w treatment */
 413                 return (DDI_SUCCESS);

 415         default:
 416                 return (DDI_FAILURE);
 417         }

 419         instance = ddi_get_instance(devi);

 421         if (ddi_soft_state_zalloc(sysctrlp, instance) != DDI_SUCCESS)
 422                 return (DDI_FAILURE);

 424         softsp = GETSOFTC(instance);

 426         /* Set the dip in the soft state */
 427         softsp->dip = devi;

 429         /* Set up the parent dip */
 430         softsp->pdip = ddi_get_parent(softsp->dip);

 432         DPRINTF(SYSCTRL_ATTACH_DEBUG, ("sysctrl: devi= 0x%p\n, softsp=0x%p\n",
 433             (void *)devi, (void *)softsp));

 435         /* First set all of the timeout values */
 436         spur_timeout_hz = drv_usectohz(SPUR_TIMEOUT_USEC);
 437         spur_long_timeout_hz = drv_usectohz(SPUR_LONG_TIMEOUT_USEC);
 438         ac_timeout_hz = drv_usectohz(AC_TIMEOUT_USEC);
 439         ps_fail_timeout_hz = drv_usectohz(PS_FAIL_TIMEOUT_USEC);
 440         pps_fan_timeout_hz = drv_usectohz(PPS_FAN_TIMEOUT_USEC);
 441         bd_insert_delay_hz = drv_usectohz(BRD_INSERT_DELAY_USEC);
 442         bd_insert_retry_hz = drv_usectohz(BRD_INSERT_RETRY_USEC);
 443         bd_remove_timeout_hz = drv_usectohz(BRD_REMOVE_TIMEOUT_USEC);
 444         blink_led_timeout_hz = drv_usectohz(BLINK_LED_TIMEOUT_USEC);
 445         overtemp_timeout_hz = drv_sectohz(OVERTEMP_TIMEOUT_SEC);
 445         overtemp_timeout_hz = drv_usectohz(OVERTEMP_TIMEOUT_SEC * MICROSEC);
 446         keyswitch_timeout_hz = drv_usectohz(KEYSWITCH_TIMEOUT_USEC);

 448         /*
```



```
 449          * Map in the registers sets that OBP hands us. According
 450          * to the sun4u device tree spec., the register sets are as
 451          * follows:
 452          *
 453          *      0       Clock Frequency Registers (contains the bit
 454          *              for enabling the remote console reset)
 455          *      1       Misc (has all the registers that we need
 456          *      2       Clock Version Register
 457          */
 458         if (ddi_map_regs(softsp->dip, 0,
 459             (caddr_t *)&softsp->clk_freq1, 0, 0)) {
 460                 cmn_err(CE_WARN, "sysctrl%d: unable to map clock frequency "
 461                     "registers", instance);
 462                 goto bad0;
 463         }

 465         if (ddi_map_regs(softsp->dip, 1,
 466             (caddr_t *)&softsp->csr, 0, 0)) {
 467                 cmn_err(CE_WARN, "sysctrl%d: unable to map internal"
 468                     "registers", instance);
 469                 goto bad1;
 470         }

 472         /*
 473          * There is a new register for newer vintage clock board nodes,
 474          * OBP register set 2 in the clock board node.
 475          *
 476          */
 477         (void) ddi_map_regs(softsp->dip, 2, (caddr_t *)&softsp->clk_ver, 0, 0);

 479         /*
 480          * Fill in the virtual addresses of the registers in the
 481          * sysctrl_soft_state structure. We do not want to calculate
 482          * them on the fly. This way we waste a little memory, but
 483          * avoid bugs down the road.
 484          */
 485         softsp->clk_freq2 = (uchar_t *)((caddr_t)softsp->clk_freq1 +
 486             SYS_OFF_CLK_FREQ2);

 488         softsp->status1 = (uchar_t *)((caddr_t)softsp->csr +
 489             SYS_OFF_STAT1);

 491         softsp->status2 = (uchar_t *)((caddr_t)softsp->csr +
 492             SYS_OFF_STAT2);

 494         softsp->ps_stat = (uchar_t *)((caddr_t)softsp->csr +
 495             SYS_OFF_PSSTAT);

 497         softsp->ps_pres = (uchar_t *)((caddr_t)softsp->csr +
 498             SYS_OFF_PSPRES);

 500         softsp->ppsr = (uchar_t *)((caddr_t)softsp->csr +
 501             SYS_OFF_PPPSR);

 503         softsp->temp_reg = (uchar_t *)((caddr_t)softsp->csr +
 504             SYS_OFF_TEMP);

 506         set_clockbrd_info();

 508         /*
 509          * Enable the hardware watchdog gate on the clock board if
 510          * map_wellknown has detected that watchdog timer is available
 511          * and user wants it to be enabled.
 512          */
 513         if (watchdog_available && watchdog_enable)
 514                 *(softsp->clk_freq2) |= TOD_RESET_EN;
```

```
515         else
516                 *(softsp->clk_freq2) &= ~TOD_RESET_EN;
517
518         /* Check for inherited faults from the PROM. */
519         if (*softsp->csr & SYS_LED_MID) {
520                 reg_fault(0, FT_PROM, FT_SYSTEM);
521         }
522
523         /*
524          * calculate and cache the number of slots on this system
525          */
526         switch (SYS_TYPE(*softsp->status1)) {
527         case SYS_16_SLOT:
528                 softsp->nslots = 16;
529                 break;
530
531         case SYS_8_SLOT:
532                 softsp->nslots = 8;
533                 break;
534
535         case SYS_4_SLOT:
536                 /* check the clk_version register - if the ptr is valid */
537                 if ((softsp->clk_ver != NULL) &&
538                     (SYS_TYPE2(*softsp->clk_ver) == SYS_PLUS_SYSTEM)) {
539                         softsp->nslots = 5;
540                 } else {
541                         softsp->nslots = 4;
542                 }
543                 break;
544
545         case SYS_TESTBED:
546         default:
547                 softsp->nslots = 0;
548                 break;
549         }
550
551
552         /* create the fault list kstat */
553         create_ft_kstats(instance);
554
555         /*
556          * Do a priming read on the ADC, and throw away the first value
557          * read. This is a feature of the ADC hardware. After a power cycle
558          * it does not contains valid data until a read occurs.
559          */
560         tmp_reg = *(softsp->temp_reg);
561
562         /* Wait 30 usec for ADC hardware to stabilize. */
563         DELAY(30);
564
565         /* shut off all interrupt sources */
566         *(softsp->csr) &= ~(SYS_PPS_FAN_FAIL_EN | SYS_PS_FAIL_EN |
567             SYS_AC_PWR_FAIL_EN | SYS_SBRD_PRES_EN);
568         tmp_reg = *(softsp->csr);
569 #ifdef lint
570         tmp_reg = tmp_reg;
571 #endif
572
573         /*
574          * Now register our high interrupt with the system.
575          */
576         if (ddi_add_intr(devi, 0, &softsp->iblock,
577             &softsp->idevice, (uint_t (*)(caddr_t))nulldev, NULL) !=
578             DDI_SUCCESS)
579                 goto bad2;
```



```
581         mutex_init(&softsp->csr_mutex, NULL, MUTEX_DRIVER,
582             (void *)softsp->iblock);
583
584         ddi_remove_intr(devi, 0, softsp->iblock);
585
586         if (ddi_add_intr(devi, 0, &softsp->iblock,
587             &softsp->idevice, system_high_handler, (caddr_t)softsp) !=
588             DDI_SUCCESS)
589                 goto bad3;
590
591         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->spur_id,
592             &softsp->spur_int_c, NULL, spur_delay, (caddr_t)softsp) !=
593             DDI_SUCCESS)
594                 goto bad4;
595
596         mutex_init(&softsp->spur_int_lock, NULL, MUTEX_DRIVER,
597             (void *)softsp->spur_int_c);
598
599
600         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->spur_high_id,
601             NULL, NULL, spur_reenable, (caddr_t)softsp) != DDI_SUCCESS)
602                 goto bad5;
603
604         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->spur_long_to_id,
605             NULL, NULL, spur_clear_count, (caddr_t)softsp) != DDI_SUCCESS)
606                 goto bad6;
607
608         /*
609          * Now register low-level ac fail handler
610          */
611         if (ddi_add_softintr(devi, DDI_SOFTINT_HIGH, &softsp->ac_fail_id,
612             NULL, NULL, ac_fail_handler, (caddr_t)softsp) != DDI_SUCCESS)
613                 goto bad7;
614
615         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->ac_fail_high_id,
616             NULL, NULL, ac_fail_reenable, (caddr_t)softsp) != DDI_SUCCESS)
617                 goto bad8;
618
619         /*
620          * Now register low-level ps fail handler
621          */
622
623         if (ddi_add_softintr(devi, DDI_SOFTINT_HIGH, &softsp->ps_fail_int_id,
624             &softsp->ps_fail_c, NULL, ps_fail_int_handler, (caddr_t)softsp) !=
625             DDI_SUCCESS)
626                 goto bad9;
627
628         mutex_init(&softsp->ps_fail_lock, NULL, MUTEX_DRIVER,
629             (void *)softsp->ps_fail_c);
630
631         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->ps_fail_poll_id,
632             NULL, NULL, ps_fail_poll_handler, (caddr_t)softsp) !=
633             DDI_SUCCESS)
634                 goto bad10;
635
636         /*
637          * Now register low-level pps fan fail handler
638          */
639         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->pps_fan_id,
640             NULL, NULL, pps_fanfail_handler, (caddr_t)softsp) !=
641             DDI_SUCCESS)
642                 goto bad11;
643
644         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->pps_fan_high_id,
645             NULL, NULL, pps_fanfail_reenable, (caddr_t)softsp) !=
646             DDI_SUCCESS)
```

```
647                 goto bad12;

649         /*
650          * Based upon a check for a current share backplane, advise
651          * that system does not support hot plug
652          *
653          */
654         if ((*(softsp->pppsr) & SYS_NOT_CURRENT_S) != 0) {
655                 cmn_err(CE_NOTE, "Hot Plug not supported in this system");
656                 sysctrl_hotplug_disabled = TRUE;
657         }

659         /*
660          * If the trigger circuit is busted or the NOT_BRD_PRES line
661          * is stuck then OBP will publish this property stating that
662          * hot plug is not available.  If this happens we will complain
663          * to the console and register a system fault.  We will also
664          * not enable the board insert interrupt for this session.
665          */
666         if (ddi_prop_op(DDI_DEV_T_ANY, softsp->dip, PROP_LEN_AND_VAL_ALLOC,
667             DDI_PROP_DONTPASS, HOTPLUG_DISABLED_PROPERTY,
668             (caddr_t)&propval, &proplen) == DDI_PROP_SUCCESS) {
669                 cmn_err(CE_WARN, "Hot Plug Unavailable [%s]", propval);
670                 reg_fault(0, FT_HOT_PLUG, FT_SYSTEM);
671                 sysctrl_hotplug_disabled = TRUE;
672                 enable_sys_interrupt &= ~SYS_SBRD_PRES_EN;
673                 kmem_free(propval, proplen);
674         }

676         sysc_board_connect_supported_init();

678         fhc_bd_sc_register(sysc_policy_update, softsp);

680         sysc_slot_info(softsp->nslots, &start, &limit, &incr);

682         /* Prime the board list. */
683         fhc_bdlist_prime(start, limit, incr);

685         /*
686          * Set up a board remove timeout call.
687          */
688         (void) fhc_bdlist_lock(-1);

690         DPRINTF(SYSCTRL_ATTACH_DEBUG,
691             ("attach: start bd_remove_poll()..."));

693         bd_remove_poll(softsp);
694         fhc_bdlist_unlock();

696         /*
697          * Now register low-level board insert handler
698          */
699         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->sbrd_pres_id,
700             NULL, NULL, bd_insert_handler, (caddr_t)softsp) != DDI_SUCCESS)
701                 goto bad13;

703         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->sbrd_gone_id,
704             NULL, NULL, bd_insert_normal, (caddr_t)softsp) != DDI_SUCCESS)
705                 goto bad14;

707         /*
708          * Now register led blink handler (interrupt level)
709          */
710         if (ddi_add_softintr(devi, DDI_SOFTINT_LOW, &softsp->blink_led_id,
711             &softsp->sys_led_c, NULL, blink_led_handler, (caddr_t)softsp) !=
712             DDI_SUCCESS)
```



```
713                 goto bad15;
714         mutex_init(&softsp->sys_led_lock, NULL, MUTEX_DRIVER,
715             (void *)softsp->sys_led_c);

717         /* initialize the bit field for all pps fans to assumed good */
718         softsp->pps_fan_saved = softsp->pps_fan_external_state =
719             SYS_AC_FAN_OK | SYS_KEYSW_FAN_OK;

721         /* prime the power supply state machines */
722         if (enable_sys_interrupt & SYS_PS_FAIL_EN)
723                 ddi_trigger_softintr(softsp->ps_fail_poll_id);


726         /* kick off the OS led blinker */
727         softsp->sys_led = FALSE;
728         ddi_trigger_softintr(softsp->blink_led_id);

730         /* Now enable selected interrupt sources */
731         mutex_enter(&softsp->csr_mutex);
732         *(softsp->csr) |= enable_sys_interrupt &
733             (SYS_AC_PWR_FAIL_EN | SYS_PS_FAIL_EN |
734             SYS_PPS_FAN_FAIL_EN | SYS_SBRD_PRES_EN);
735         tmp_reg = *(softsp->csr);
736 #ifdef lint
737         tmp_reg = tmp_reg;
738 #endif
739         mutex_exit(&softsp->csr_mutex);

741         /* Initialize the temperature */
742         init_temp_arrays(&softsp->tempstat);

744         /*
745          * initialize key switch shadow state
746          */
747         softsp->key_shadow = KEY_BOOT;

749         /*
750          * Now add this soft state structure to the front of the linked list
751          * of soft state structures.
752          */
753         if (sys_list == (struct sysctrl_soft_state *)NULL) {
754                 mutex_init(&sslist_mutex, NULL, MUTEX_DEFAULT, NULL);
755         }
756         mutex_enter(&sslist_mutex);
757         softsp->next = sys_list;
758         sys_list = softsp;
759         mutex_exit(&sslist_mutex);

761         /* Setup the kstats for this device */
762         sysctrl_add_kstats(softsp);

764         /* kick off the PPS fan poll routine */
765         pps_fan_poll(softsp);

767         if (sysctrl_overtemp_thread_started == 0) {
768                 /*
769                  * set up the overtemp condition variable before
770                  * starting the thread.
771                  */
772                 cv_init(&overtemp_cv, NULL, CV_DRIVER, NULL);

774                 /*
775                  * start up the overtemp polling thread
776                  */
777                 (void) thread_create(NULL, 0, (void (*)())sysctrl_overtemp_poll,
778                     NULL, 0, &p0, TS_RUN, minclsyspri);
```

```
779                 sysctrl_overtemp_thread_started++;
780         }
782         if (sysctrl_keyswitch_thread_started == 0) {
783                 extern void (*abort_seq_handler)();
785                 /*
786                  * interpose sysctrl's abort sequence handler
787                  */
788                 abort_seq_handler = sysctrl_abort_seq_handler;
790                 /*
791                  * set up the key switch condition variable before
792                  * starting the thread
793                  */
794                 cv_init(&keyswitch_cv, NULL, CV_DRIVER, NULL);
796                 /*
797                  * start up the key switch polling thread
798                  */
799                 (void) thread_create(NULL, 0,
800                     (void (*)())sysctrl_keyswitch_poll, NULL, 0, &p0,
801                     TS_RUN, minclsyspri);
802                 sysctrl_keyswitch_thread_started++;
803         }
805         /*
806          * perform initialization to allow setting of powerfail-time
807          */
808         if ((dip = ddi_find_devinfo("options", -1, 0)) == NULL)
809                 softsp->options_nodeid = (pnode_t)NULL;
810         else
811                 softsp->options_nodeid = (pnode_t)ddi_get_nodeid(dip);
813         DPRINTF(SYSCTRL_ATTACH_DEBUG,
814             ("sysctrl: Creating devices start:%d, limit:%d, incr:%d\n",
815             start, limit, incr));
817         /*
818          * Create minor node for each system attachment points
819          */
820         for (slot_num = start; slot_num < limit; slot_num = slot_num + incr) {
821                 char name[30];
822                 (void) sprintf(name, "slot%d", slot_num);
823                 if (ddi_create_minor_node(devi, name, S_IFCHR,
824                     (PUTINSTANCE(instance) | slot_num),
825                     DDI_NT_ATTACHMENT_POINT, 0) == DDI_FAILURE) {
826                         cmn_err(CE_WARN, "sysctrl%d: \"%s\" "
827                             "ddi_create_minor_node failed",
828                             instance, name);
829                         goto bad16;
830                 }
831         }
833         ddi_report_dev(devi);
835         /*
836          * Remote console is inherited from POST
837          */
838         if ((*(softsp->clk_freq2) & RCONS_UART_EN) == 0) {
839                 softsp->enable_rcons_atboot = FALSE;
840                 cmn_err(CE_WARN, "Remote console not active");
841         } else
842                 softsp->enable_rcons_atboot = TRUE;
844         return (DDI_SUCCESS);
```



```
846 bad16:
847         cv_destroy(&keyswitch_cv);
848         cv_destroy(&overtemp_cv);
849         mutex_destroy(&sslist_mutex);
850         mutex_destroy(&softsp->sys_led_lock);
851         ddi_remove_softintr(softsp->blink_led_id);
852 bad15:
853         ddi_remove_softintr(softsp->sbrd_gone_id);
854 bad14:
855         ddi_remove_softintr(softsp->sbrd_pres_id);
856 bad13:
857         ddi_remove_softintr(softsp->pps_fan_high_id);
858 bad12:
859         ddi_remove_softintr(softsp->pps_fan_id);
860 bad11:
861         ddi_remove_softintr(softsp->ps_fail_poll_id);
862 bad10:
863         mutex_destroy(&softsp->ps_fail_lock);
864         ddi_remove_softintr(softsp->ps_fail_int_id);
865 bad9:
866         ddi_remove_softintr(softsp->ac_fail_high_id);
867 bad8:
868         ddi_remove_softintr(softsp->ac_fail_id);
869 bad7:
870         ddi_remove_softintr(softsp->spur_long_to_id);
871 bad6:
872         ddi_remove_softintr(softsp->spur_high_id);
873 bad5:
874         mutex_destroy(&softsp->spur_int_lock);
875         ddi_remove_softintr(softsp->spur_id);
876 bad4:
877         ddi_remove_intr(devi, 0, softsp->iblock);
878 bad3:
879         mutex_destroy(&softsp->csr_mutex);
880 bad2:
881         ddi_unmap_regs(softsp->dip, 1, (caddr_t *)&softsp->csr, 0, 0);
882         if (softsp->clk_ver != NULL)
883                 ddi_unmap_regs(softsp->dip, 2, (caddr_t *)&softsp->clk_ver,
884                     0, 0);
885 bad1:
886         ddi_unmap_regs(softsp->dip, 0, (caddr_t *)&softsp->clk_freq1, 0, 0);
888 bad0:
889         ddi_soft_state_free(sysctrlp, instance);
890         ddi_remove_minor_node(dip, NULL);
891         cmn_err(CE_WARN,
892             "sysctrl%d: Initialization failure. Some system level events,"
893             " {AC Fail, Fan Failure, PS Failure} not detected", instance);
894         return (DDI_FAILURE);
895 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18720 Wed Aug 19 07:25:31 2015
new/usr/src/uts/sun4v/io/platsvc.c
XXXX introduce drv_sectohz
**********************************************************
```

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

  22 /*
  23  * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 /*
  27  * sun4v Platform Services Module
  28  */

  30 #include <sys/modctl.h>
  31 #include <sys/cmn_err.h>
  32 #include <sys/machsystm.h>
  33 #include <sys/note.h>
  34 #include <sys/uadmin.h>
  35 #include <sys/ds.h>
  36 #include <sys/platsvc.h>
  37 #include <sys/ddi.h>
  38 #include <sys/suspend.h>
  39 #include <sys/proc.h>
  40 #include <sys/disp.h>
  41 #include <sys/drctl.h>

  43 /*
  44  * Debugging routines
  45  */
  46 #ifdef DEBUG
  47 uint_t ps_debug = 0x0;
  48 #define DBG     if (ps_debug) printf
  49 #else /* DEBUG */
  50 #define DBG     _NOTE(CONSTCOND) if (0) printf
  51 #endif /* DEBUG */

  53 /*
  54  * Time resolution conversions.
  55  */
  56 #define MS2NANO(x)      ((x) * MICROSEC)
  57 #define MS2SEC(x)       ((x) / MILLISEC)
  58 #define MS2MIN(x)       (MS2SEC(x) / 60)
  59 #define SEC2HZ(x)       drv_sectohz(x)
  59 #define SEC2HZ(x)       (drv_usectohz((x) * MICROSEC))
```



```
  61 /*
  62  * Domains Services interaction
  63  */
  64 static ds_svc_hdl_t     ds_md_handle;
  65 static ds_svc_hdl_t     ds_shutdown_handle;
  66 static ds_svc_hdl_t     ds_panic_handle;
  67 static ds_svc_hdl_t     ds_suspend_handle;

  69 static ds_ver_t         ps_vers[] = {{ 1, 0 }};
  70 #define PS_NVERS        (sizeof (ps_vers) / sizeof (ps_vers[0]))

  72 static ds_capability_t ps_md_cap = {
  73         "md-update",            /* svc_id */
  74         ps_vers,                /* vers */
  75         PS_NVERS                /* nvers */
  76 };
______unchanged_portion_omitted_
```

```
**********************************************************
  222672 Wed Aug 19 07:25:31 2015
new/usr/src/uts/sun4v/io/vdc.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

4400 /* ----------------------------------------------------------------------- */

4402 /*
4403  * The following functions process the incoming messages from vds
4404  */

4406 /*
4407  * Function:
4408  *      vdc_process_msg_thread()
4409  *
4410  * Description:
4411  *
4412  *      Main VDC message processing thread. Each vDisk instance
4413  *      consists of a copy of this thread. This thread triggers
4414  *      all the handshakes and data exchange with the server. It
4415  *      also handles all channel resets
4416  *
4417  * Arguments:
4418  *      vdc     - soft state pointer for this instance of the device driver.
4419  *
4420  * Return Code:
4421  *      None
4422  */
4423 static void
4424 vdc_process_msg_thread(vdc_t *vdcp)
4425 {
4426         boolean_t       failure_msg = B_FALSE;
4427         int             status;
4428         int             ctimeout;
4429         timeout_id_t    tmid = 0;
4430         clock_t         ldcup_timeout = 0;
4431         vdc_server_t    *srvr;
4432         vdc_service_state_t svc_state;
4433         int             hshake_cnt = 0;
4434         int             hattr_cnt = 0;

4436         mutex_enter(&vdcp->lock);

4438         ASSERT(vdcp->lifecycle == VDC_LC_ATTACHING);

4440         for (;;) {

4442 #define Q(_s)   (vdcp->state == _s) ? #_s :
4443                 DMSG(vdcp, 3, "state = %d (%s)\n", vdcp->state,
4444                     Q(VDC_STATE_INIT)
4445                     Q(VDC_STATE_INIT_WAITING)
4446                     Q(VDC_STATE_NEGOTIATE)
4447                     Q(VDC_STATE_HANDLE_PENDING)
4448                     Q(VDC_STATE_FAULTED)
4449                     Q(VDC_STATE_FAILED)
4450                     Q(VDC_STATE_RUNNING)
4451                     Q(VDC_STATE_RESETTING)
4452                     Q(VDC_STATE_DETACH)
4453                     "UNKNOWN");
4454 #undef Q

4456                 switch (vdcp->state) {
4457                 case VDC_STATE_INIT:
```



```
4459                         /*
4460                          * If requested, start a timeout to check if the
4461                          * connection with vds is established in the
4462                          * specified delay. If the timeout expires, we
4463                          * will cancel any pending request.
4464                          *
4465                          * If some reset have occurred while establishing
4466                          * the connection, we already have a timeout armed
4467                          * and in that case we don't need to arm a new one.
4468                          *
4469                          * The same rule applies when there are multiple vds's.
4470                          * If either a connection cannot be established or
4471                          * the handshake times out, the connection thread will
4472                          * try another server. The 'ctimeout' will report
4473                          * back an error after it expires irrespective of
4474                          * whether the vdisk is trying to connect to just
4475                          * one or multiple servers.
4476                          */
4477                         ctimeout = (vdc_timeout != 0)?
4478                             vdc_timeout : vdcp->curr_server->ctimeout;

4480                         if (ctimeout != 0 && tmid == 0) {
4481                                 tmid = timeout(vdc_connection_timeout, vdcp,
4482                                     drv_sectohz(ctimeout));
4482                                     ctimeout * drv_usectohz(MICROSEC));
4483                         }

4485                         /* Switch to STATE_DETACH if drv is detaching */
4486                         if (vdcp->lifecycle == VDC_LC_DETACHING) {
4487                                 vdcp->state = VDC_STATE_DETACH;
4488                                 break;
4489                         }

4491                         /* Check if the timeout has been reached */
4492                         if (vdcp->ctimeout_reached) {
4493                                 ASSERT(tmid != 0);
4494                                 tmid = 0;
4495                                 vdcp->state = VDC_STATE_FAILED;
4496                                 break;
4497                         }

4499                         /*
4500                          * Switch to another server when we reach the limit of
4501                          * the number of handshake per server or if we have done
4502                          * an attribute negotiation.
4503                          */
4504                         if (hshake_cnt >= vdc_hshake_retries || hattr_cnt > 0) {

4506                                 if (!vdc_handshake_retry(vdcp, hshake_cnt,
4507                                     hattr_cnt)) {
4508                                         DMSG(vdcp, 0, "[%d] too many "
4509                                             "handshakes", vdcp->instance);
4510                                         vdcp->state = VDC_STATE_FAILED;
4511                                         break;
4512                                 }

4514                                 vdc_switch_server(vdcp);

4516                                 hshake_cnt = 0;
4517                                 hattr_cnt = 0;
4518                         }

4520                         hshake_cnt++;

4522                         /* Bring up connection with vds via LDC */
4523                         status = vdc_start_ldc_connection(vdcp);
```

```
4524                        if (status != EINVAL) {
4525                                vdcp->state = VDC_STATE_INIT_WAITING;
4526                        } else {
4527                                vdcp->curr_server->svc_state =
4528                                    VDC_SERVICE_FAILED;
4529                                vdc_print_svc_status(vdcp);
4530                        }
4531                        break;

4533                case VDC_STATE_INIT_WAITING:

4535                        /* if channel is UP, start negotiation */
4536                        if (vdcp->curr_server->ldc_state == LDC_UP) {
4537                                vdcp->state = VDC_STATE_NEGOTIATE;
4538                                break;
4539                        }

4541                        /*
4542                         * Wait for LDC_UP. If it times out and we have multiple
4543                         * servers then we will retry using a different server.
4544                         */
4545                        ldcup_timeout = ddi_get_lbolt() + drv_sectohz(vdc_ldcup_
4545                        ldcup_timeout = ddi_get_lbolt() + (vdc_ldcup_timeout *
4546                            drv_usectohz(MICROSEC));
4546                        status = cv_timedwait(&vdcp->initwait_cv, &vdcp->lock,
4547                            ldcup_timeout);
4548                        if (status == -1 &&
4549                            vdcp->state == VDC_STATE_INIT_WAITING &&
4550                            vdcp->curr_server->ldc_state != LDC_UP) {
4551                                /* timed out & still waiting */
4552                                vdcp->curr_server->svc_state =
4553                                    VDC_SERVICE_FAILED;
4554                                vdc_print_svc_status(vdcp);
4555                                vdcp->state = VDC_STATE_INIT;
4556                                break;
4557                        }

4559                        if (vdcp->state != VDC_STATE_INIT_WAITING) {
4560                                DMSG(vdcp, 0,
4561                                    "state moved to %d out from under us...\n",
4562                                    vdcp->state);
4563                        }
4564                        break;

4566                case VDC_STATE_NEGOTIATE:
4567                        switch (status = vdc_ver_negotiation(vdcp)) {
4568                        case 0:
4569                                break;
4570                        default:
4571                                DMSG(vdcp, 0, "ver negotiate failed (%d)..\n",
4572                                    status);
4573                                goto reset;
4574                        }

4576                        hattr_cnt++;

4578                        switch (status = vdc_attr_negotiation(vdcp)) {
4579                        case 0:
4580                                break;
4581                        default:
4582                                DMSG(vdcp, 0, "attr negotiate failed (%d)..\n",
4583                                    status);
4584                                goto reset;
4585                        }

4587                        switch (status = vdc_dring_negotiation(vdcp)) {
```



```
4588                        case 0:
4589                                break;
4590                        default:
4591                                DMSG(vdcp, 0, "dring negotiate failed (%d)..\n",
4592                                    status);
4593                                goto reset;
4594                        }

4596                        switch (status = vdc_rdx_exchange(vdcp)) {
4597                        case 0:
4598                                vdcp->state = VDC_STATE_HANDLE_PENDING;
4599                                goto done;
4600                        default:
4601                                DMSG(vdcp, 0, "RDX xchg failed ..(%d)\n",
4602                                    status);
4603                                goto reset;
4604                        }
4605 reset:
4606                        DMSG(vdcp, 0, "negotiation failed: resetting (%d)\n",
4607                            status);
4608                        vdcp->state = VDC_STATE_RESETTING;
4609                        vdcp->self_reset = B_TRUE;
4610                        vdcp->curr_server->svc_state = VDC_SERVICE_FAILED;
4611                        vdc_print_svc_status(vdcp);
4612 done:
4613                        DMSG(vdcp, 0, "negotiation complete (state=0x%x)...\n",
4614                            vdcp->state);
4615                        break;

4617                case VDC_STATE_HANDLE_PENDING:

4619                        DMSG(vdcp, 0, "[%d] connection to service domain is up",
4620                            vdcp->instance);
4621                        vdcp->curr_server->svc_state = VDC_SERVICE_CONNECTED;

4623                        mutex_exit(&vdcp->lock);

4625                        /*
4626                         * If we have multiple servers, check that the backend
4627                         * is effectively available before resubmitting any IO.
4628                         */
4629                        if (vdcp->num_servers > 1 &&
4630                            vdc_eio_check(vdcp, 0) != 0) {
4631                                mutex_enter(&vdcp->lock);
4632                                vdcp->curr_server->svc_state =
4633                                    VDC_SERVICE_FAULTED;
4634                                vdcp->state = VDC_STATE_FAULTED;
4635                                break;
4636                        }

4638                        if (tmid != 0) {
4639                                (void) untimeout(tmid);
4640                                tmid = 0;
4641                                vdcp->ctimeout_reached = B_FALSE;
4642                        }

4644                        /*
4645                         * Setup devid
4646                         */
4647                        (void) vdc_setup_devid(vdcp);

4649                        status = vdc_resubmit_backup_dring(vdcp);

4651                        mutex_enter(&vdcp->lock);

4653                        if (status) {
```

```
4654                                vdcp->state = VDC_STATE_RESETTING;
4655                                vdcp->self_reset = B_TRUE;
4656                                vdcp->curr_server->svc_state =
4657                                    VDC_SERVICE_FAILED;
4658                                vdc_print_svc_status(vdcp);
4659                        } else {
4660                                vdcp->state = VDC_STATE_RUNNING;
4661                        }
4662                        break;

4664                case VDC_STATE_FAULTED:
4665                        /*
4666                         * Server is faulted because the backend is unavailable.
4667                         * If all servers are faulted then we mark the service
4668                         * as failed, otherwise we reset to switch to another
4669                         * server.
4670                         */
4671                        vdc_print_svc_status(vdcp);

4673                        /* check if all servers are faulted */
4674                        for (srvr = vdcp->server_list; srvr != NULL;
4675                            srvr = srvr->next) {
4676                                svc_state = srvr->svc_state;
4677                                if (svc_state != VDC_SERVICE_FAULTED)
4678                                        break;
4679                        }

4681                        if (srvr != NULL) {
4682                                vdcp->state = VDC_STATE_RESETTING;
4683                                vdcp->self_reset = B_TRUE;
4684                        } else {
4685                                vdcp->state = VDC_STATE_FAILED;
4686                        }
4687                        break;

4689                case VDC_STATE_FAILED:
4690                        /*
4691                         * We reach this state when we are unable to access the
4692                         * backend from any server, either because of a maximum
4693                         * connection retries or timeout, or because the backend
4694                         * is unavailable.
4695                         *
4696                         * Then we cancel the backup DRing so that errors get
4697                         * reported and we wait for a new I/O before attempting
4698                         * another connection.
4699                         */

4701                        cmn_err(CE_NOTE, "vdisk@%d disk access failed",
4702                            vdcp->instance);
4703                        failure_msg = B_TRUE;

4705                        if (vdcp->lifecycle == VDC_LC_ATTACHING) {
4706                                vdcp->lifecycle = VDC_LC_ONLINE_PENDING;
4707                                vdcp->hattr_min = vdc_hattr_min_initial;
4708                        } else {
4709                                vdcp->hattr_min = vdc_hattr_min;
4710                        }

4712                        /* cancel any timeout */
4713                        if (tmid != 0) {
4714                                (void) untimeout(tmid);
4715                                tmid = 0;
4716                        }

4718                        /* cancel pending I/Os */
4719                        cv_broadcast(&vdcp->running_cv);
```



```
4720                        vdc_cancel_backup_dring(vdcp);

4722                        /* wait for new I/O */
4723                        while (!vdcp->io_pending)
4724                                cv_wait(&vdcp->io_pending_cv, &vdcp->lock);

4726                        /*
4727                         * There's a new IO pending. Try to re-establish a
4728                         * connection. Mark all services as offline, so that
4729                         * we don't stop again before having retried all
4730                         * servers.
4731                         */
4732                        for (srvr = vdcp->server_list; srvr != NULL;
4733                            srvr = srvr->next) {
4734                                srvr->svc_state = VDC_SERVICE_OFFLINE;
4735                                srvr->hshake_cnt = 0;
4736                                srvr->hattr_cnt = 0;
4737                                srvr->hattr_total = 0;
4738                        }

4740                        /* reset variables */
4741                        hshake_cnt = 0;
4742                        hattr_cnt = 0;
4743                        vdcp->ctimeout_reached = B_FALSE;

4745                        vdcp->state = VDC_STATE_RESETTING;
4746                        vdcp->self_reset = B_TRUE;
4747                        break;

4749                /* enter running state */
4750                case VDC_STATE_RUNNING:

4752                        if (vdcp->lifecycle == VDC_LC_DETACHING) {
4753                                vdcp->state = VDC_STATE_DETACH;
4754                                break;
4755                        }

4757                        vdcp->lifecycle = VDC_LC_ONLINE;

4759                        if (failure_msg) {
4760                                cmn_err(CE_NOTE, "vdisk@%d disk access "
4761                                    "recovered", vdcp->instance);
4762                                failure_msg = B_FALSE;
4763                        }

4765                        /*
4766                         * Signal anyone waiting for the connection
4767                         * to come on line.
4768                         */
4769                        cv_broadcast(&vdcp->running_cv);

4771                        /* backend has to be checked after reset */
4772                        if (vdcp->failfast_interval != 0 ||
4773                            vdcp->num_servers > 1)
4774                                cv_signal(&vdcp->eio_cv);

4776                        /* ownership is lost during reset */
4777                        if (vdcp->ownership & VDC_OWNERSHIP_WANTED)
4778                                vdcp->ownership |= VDC_OWNERSHIP_RESET;
4779                        cv_signal(&vdcp->ownership_cv);

4781                        vdcp->curr_server->svc_state = VDC_SERVICE_ONLINE;
4782                        vdc_print_svc_status(vdcp);

4784                        mutex_exit(&vdcp->lock);
```

```
4786                        for (;;) {
4787                                vio_msg_t msg;
4788                                status = vdc_wait_for_response(vdcp, &msg);
4789                                if (status) break;

4791                                DMSG(vdcp, 1, "[%d] new pkt(s) available\n",
4792                                    vdcp->instance);
4793                                status = vdc_process_data_msg(vdcp, &msg);
4794                                if (status) {
4795                                        DMSG(vdcp, 1, "[%d] process_data_msg "
4796                                            "returned err=%d\n", vdcp->instance,
4797                                            status);
4798                                        break;
4799                                }

4801                        }

4803                        mutex_enter(&vdcp->lock);

4805                        /* all servers are now offline */
4806                        for (srvr = vdcp->server_list; srvr != NULL;
4807                            srvr = srvr->next) {
4808                                srvr->svc_state = VDC_SERVICE_OFFLINE;
4809                                srvr->log_state = VDC_SERVICE_NONE;
4810                                srvr->hshake_cnt = 0;
4811                                srvr->hattr_cnt = 0;
4812                                srvr->hattr_total = 0;
4813                        }

4815                        hshake_cnt = 0;
4816                        hattr_cnt = 0;

4818                        vdc_print_svc_status(vdcp);

4820                        vdcp->state = VDC_STATE_RESETTING;
4821                        vdcp->self_reset = B_TRUE;
4822                        break;

4824                case VDC_STATE_RESETTING:
4825                        /*
4826                         * When we reach this state, we either come from the
4827                         * VDC_STATE_RUNNING state and we can have pending
4828                         * request but no timeout is armed; or we come from
4829                         * the VDC_STATE_INIT_WAITING, VDC_NEGOTIATE or
4830                         * VDC_HANDLE_PENDING state and there is no pending
4831                         * request or pending requests have already been copied
4832                         * into the backup dring. So we can safely keep the
4833                         * connection timeout armed while we are in this state.
4834                         */

4836                        DMSG(vdcp, 0, "Initiating channel reset "
4837                            "(pending = %d)\n", (int)vdcp->threads_pending);

4839                        if (vdcp->self_reset) {
4840                                DMSG(vdcp, 0,
4841                                    "[%d] calling stop_ldc_connection.\n",
4842                                    vdcp->instance);
4843                                status = vdc_stop_ldc_connection(vdcp);
4844                                vdcp->self_reset = B_FALSE;
4845                        }

4847                        /*
4848                         * Wait for all threads currently waiting
4849                         * for a free dring entry to use.
4850                         */
4851                        while (vdcp->threads_pending) {
```



```
4852                                cv_broadcast(&vdcp->membind_cv);
4853                                cv_broadcast(&vdcp->dring_free_cv);
4854                                mutex_exit(&vdcp->lock);
4855                                /* give the waiters enough time to wake up */
4856                                delay(vdc_hz_min_ldc_delay);
4857                                mutex_enter(&vdcp->lock);
4858                        }

4860                        ASSERT(vdcp->threads_pending == 0);

4862                        /* Sanity check that no thread is receiving */
4863                        ASSERT(vdcp->read_state != VDC_READ_WAITING);

4865                        vdcp->read_state = VDC_READ_IDLE;
4866                        vdcp->io_pending = B_FALSE;

4868                        /*
4869                         * Cleanup any pending eio. These I/Os are going to
4870                         * be resubmitted.
4871                         */
4872                        vdc_eio_unqueue(vdcp, 0, B_FALSE);

4874                        vdc_backup_local_dring(vdcp);

4876                        /* cleanup the old d-ring */
4877                        vdc_destroy_descriptor_ring(vdcp);

4879                        /* go and start again */
4880                        vdcp->state = VDC_STATE_INIT;

4882                        break;

4884                case VDC_STATE_DETACH:
4885                        DMSG(vdcp, 0, "[%d] Reset thread exit cleanup ..\n",
4886                            vdcp->instance);

4888                        /* cancel any pending timeout */
4889                        mutex_exit(&vdcp->lock);
4890                        if (tmid != 0) {
4891                                (void) untimeout(tmid);
4892                                tmid = 0;
4893                        }
4894                        mutex_enter(&vdcp->lock);

4896                        /*
4897                         * Signal anyone waiting for connection
4898                         * to come online
4899                         */
4900                        cv_broadcast(&vdcp->running_cv);

4902                        while (vdcp->sync_op_cnt > 0) {
4903                                cv_broadcast(&vdcp->sync_blocked_cv);
4904                                mutex_exit(&vdcp->lock);
4905                                /* give the waiters enough time to wake up */
4906                                delay(vdc_hz_min_ldc_delay);
4907                                mutex_enter(&vdcp->lock);
4908                        }

4910                        mutex_exit(&vdcp->lock);

4912                        DMSG(vdcp, 0, "[%d] Msg processing thread exiting ..\n",
4913                            vdcp->instance);
4914                        thread_exit();
4915                        break;
4916                }
4917        }
```

```
4918 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  150355 Wed Aug 19 07:25:31 2015
new/usr/src/uts/sun4v/io/vnet_gen.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

4100 /*
4101  * Initiate handshake with the peer by sending various messages
4102  * based on the handshake-phase that the channel is currently in.
4103  */
4104 static int
4105 vgen_handshake(vgen_ldc_t *ldcp)
4106 {
4107         uint32_t        hphase = ldcp->hphase;
4108         vgen_t          *vgenp = LDC_TO_VGEN(ldcp);
4109         int             rv = 0;
4110         timeout_id_t    htid;

4112         switch (hphase) {

4114         case VH_PHASE1:

4116                 /*
4117                  * start timer, for entire handshake process, turn this timer
4118                  * off if all phases of handshake complete successfully and
4119                  * hphase goes to VH_DONE(below) or channel is reset due to
4120                  * errors or vgen_ldc_uninit() is invoked(vgen_stop).
4121                  */
4122                 ASSERT(ldcp->htid == 0);
4123                 ldcp->htid = timeout(vgen_hwatchdog, (caddr_t)ldcp,
4124                     drv_sectohz(vgen_hwd_interval));
4124                     drv_usectohz(vgen_hwd_interval * MICROSEC));

4126                 /* Phase 1 involves negotiating the version */
4127                 rv = vgen_send_version_negotiate(ldcp);
4128                 break;

4130         case VH_PHASE2:
4131                 rv = vgen_handshake_phase2(ldcp);
4132                 break;

4134         case VH_PHASE3:
4135                 rv = vgen_handshake_phase3(ldcp);
4136                 break;

4138         case VH_PHASE4:
4139                 rv = vgen_send_rdx_info(ldcp);
4140                 break;

4142         case VH_DONE:

4144                 ldcp->ldc_reset_count = 0;

4146                 DBG1(vgenp, ldcp, "Handshake Done\n");

4148                 /*
4149                  * The channel is up and handshake is done successfully. Now we
4150                  * can mark the channel link_state as 'up'. We also notify the
4151                  * stack if the channel is connected to vswitch.
4152                  */
4153                 ldcp->link_state = LINK_STATE_UP;

4155                 if (ldcp->portp == vgenp->vsw_portp) {
4156                         /*
4157                          * If this channel(port) is connected to vsw,
```



```
4158                          * need to sync multicast table with vsw.
4159                          */
4160                         rv = vgen_send_mcast_info(ldcp);
4161                         if (rv != VGEN_SUCCESS)
4162                                 break;

4164                         if (vgenp->pls_negotiated == B_FALSE) {
4165                                 /*
4166                                  * We haven't negotiated with vswitch to get
4167                                  * physical link state updates. We can update
4168                                  * update the stack at this point as the
4169                                  * channel to vswitch is up and the handshake
4170                                  * is done successfully.
4171                                  *
4172                                  * If we have negotiated to get physical link
4173                                  * state updates, then we won't notify the
4174                                  * the stack here; we do that as soon as
4175                                  * vswitch sends us the initial phys link state
4176                                  * (see vgen_handle_physlink_info()).
4177                                  */
4178                                 mutex_exit(&ldcp->cblock);
4179                                 vgen_link_update(vgenp, ldcp->link_state);
4180                                 mutex_enter(&ldcp->cblock);
4181                         }
4182                 }

4184                 if (ldcp->htid != 0) {
4185                         htid = ldcp->htid;
4186                         ldcp->htid = 0;

4188                         mutex_exit(&ldcp->cblock);
4189                         (void) untimeout(htid);
4190                         mutex_enter(&ldcp->cblock);
4191                 }

4193                 /*
4194                  * Check if mac layer should be notified to restart
4195                  * transmissions. This can happen if the channel got
4196                  * reset and while tx_blocked is set.
4197                  */
4198                 mutex_enter(&ldcp->tclock);
4199                 if (ldcp->tx_blocked) {
4200                         vio_net_tx_update_t vtx_update =
4201                             ldcp->portp->vcb.vio_net_tx_update;

4203                         ldcp->tx_blocked = B_FALSE;
4204                         vtx_update(ldcp->portp->vhp);
4205                 }
4206                 mutex_exit(&ldcp->tclock);

4208                 /* start transmit watchdog timer */
4209                 ldcp->wd_tid = timeout(vgen_tx_watchdog, (caddr_t)ldcp,
4210                     drv_usectohz(vgen_txwd_interval * 1000));

4212                 break;

4214         default:
4215                 break;
4216         }

4218         return (rv);
4219 }
______unchanged_portion_omitted_
```

```
**********************************************************
   48937 Wed Aug 19 07:25:32 2015
new/usr/src/uts/sun4v/io/vsw_switching.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 163 /*
 164  * Thread to setup switching mode. This thread is created during vsw_attach()
 165  * initially. It invokes vsw_setup_switching() and keeps retrying while the
 166  * returned value is EAGAIN. The thread exits when the switching mode setup is
 167  * done successfully or when the error returned is not EAGAIN. This thread may
 168  * also get created from vsw_update_md_prop() if the switching mode needs to be
 169  * updated.
 170  */
 171 void
 172 vsw_setup_switching_thread(void *arg)
 173 {
 174         callb_cpr_t     cprinfo;
 175         vsw_t           *vswp =  (vsw_t *)arg;
 176         clock_t         wait_time;
 177         clock_t         xwait;
 178         clock_t         wait_rv;
 179         int             rv;

 181         /* wait time used on successive retries */
 182         xwait = drv_sectohz(vsw_setup_switching_delay);
 182         xwait = drv_usectohz(vsw_setup_switching_delay * MICROSEC);

 184         CALLB_CPR_INIT(&cprinfo, &vswp->sw_thr_lock, callb_generic_cpr,
 185             "vsw_setup_sw_thread");

 187         mutex_enter(&vswp->sw_thr_lock);

 189         while ((vswp->sw_thr_flags & VSW_SWTHR_STOP) == 0) {

 191                 CALLB_CPR_SAFE_BEGIN(&cprinfo);

 193                 /* Wait for sometime before (re)trying setup_switching() */
 194                 wait_time = ddi_get_lbolt() + xwait;
 195                 while ((vswp->sw_thr_flags & VSW_SWTHR_STOP) == 0) {
 196                         wait_rv = cv_timedwait(&vswp->sw_thr_cv,
 197                             &vswp->sw_thr_lock, wait_time);
 198                         if (wait_rv == -1) {    /* timed out */
 199                                 break;
 200                         }
 201                 }

 203                 CALLB_CPR_SAFE_END(&cprinfo, &vswp->sw_thr_lock)

 205                 if ((vswp->sw_thr_flags & VSW_SWTHR_STOP) != 0) {
 206                         /*
 207                          * If there is a stop request, process that first and
 208                          * exit the loop. Continue to hold the mutex which gets
 209                          * released in CALLB_CPR_EXIT().
 210                          */
 211                         break;
 212                 }

 214                 mutex_exit(&vswp->sw_thr_lock);
 215                 rv = vsw_setup_switching(vswp);
 216                 if (rv == 0) {
 217                         vsw_setup_switching_post_process(vswp);
 218                 }
 219                 mutex_enter(&vswp->sw_thr_lock);
 220                 if (rv != EAGAIN) {
```



```
 221                         break;
 222                 }

 224         }

 226         vswp->sw_thr_flags &= ~VSW_SWTHR_STOP;
 227         vswp->sw_thread = NULL;
 228         CALLB_CPR_EXIT(&cprinfo);
 229         thread_exit();
 230 }
______unchanged_portion_omitted_
```

```
**********************************************************
   39076 Wed Aug 19 07:25:32 2015
new/usr/src/uts/sun4v/ontario/io/tsalarm.c
XXXX introduce drv_sectohz
**********************************************************
______unchanged_portion_omitted_

 730 static int
 731 glvc_alarm_get(int alarm_type, int *alarm_state, tsalarm_softc_t *sc)
 732 {
 733         tsal_pcp_alarm_req_t    *req_ptr = NULL;
 734         tsal_pcp_alarm_resp_t   *resp_ptr = NULL;
 735         tsal_pcp_msg_t          send_msg;
 736         tsal_pcp_msg_t          recv_msg;
 737         int                     status = -1;

 739         /*
 740          * setup the request data to attach to the libpcp msg
 741          */
 742         if (sc->req_ptr == NULL) {
 743                 goto alarm_return;
 744         }

 746         req_ptr = sc->req_ptr;

 748         req_ptr->alarm_action = PCP_ALARM_STATUS;
 749         req_ptr->alarm_id = alarm_type;

 751         send_msg.msg_type = PCP_ALARM_CONTROL;
 752         send_msg.sub_type = NULL;
 753         send_msg.msg_len = sizeof (tsal_pcp_alarm_req_t);
 754         send_msg.msg_data = (uint8_t *)req_ptr;

 756         /*
 757          * send the request, receive the response
 758          */
 759         if (tsal_pcp_send_recv(sc, &send_msg, &recv_msg,
 760             PCP_COMM_TIMEOUT) < 0) {
 761                 /* we either timed out or erred; either way try again */
 762                 (void) delay(drv_sectohz(PCP_COMM_TIMEOUT));
 762                 (void) delay(PCP_COMM_TIMEOUT * drv_usectohz(1000000));

 764                 if (tsal_pcp_send_recv(sc, &send_msg, &recv_msg,
 765                     PCP_COMM_TIMEOUT) < 0) {
 766                         cmn_err(CE_WARN, "tsalarm: communication failure");
 767                         goto alarm_return;
 768                 }
 769         }

 771         /*
 772          * validate that this data was meant for us
 773          */
 774         if (recv_msg.msg_type != PCP_ALARM_CONTROL_R) {
 775                 cmn_err(CE_WARN, "tsalarm: unbound packet received");
 776                 goto alarm_return;
 777         }

 779         /*
 780          * verify that the Alarm action has taken place
 781          */
 782         resp_ptr = (tsal_pcp_alarm_resp_t *)recv_msg.msg_data;
 783         if (resp_ptr->status == PCP_ALARM_ERROR) {
 784                 cmn_err(CE_WARN, "tsalarm: failed to get alarm status");
 785                 goto alarm_return;
 786         }
```



```
 788         if (resp_ptr->alarm_state == ALARM_STATE_UNKNOWN)
 789                 cmn_err(CE_WARN, "tsalarm: ALARM set to unknown state");

 791         *alarm_state = resp_ptr->alarm_state;
 792         status = TSAL_PCP_OK;

 794 alarm_return:
 795         return (status);
 796 }

 798 static int
 799 glvc_alarm_set(int alarm_type, int new_state, tsalarm_softc_t *sc)
 800 {
 801         tsal_pcp_alarm_req_t    *req_ptr = NULL;
 802         tsal_pcp_alarm_resp_t   *resp_ptr = NULL;
 803         tsal_pcp_msg_t          send_msg;
 804         tsal_pcp_msg_t          recv_msg;
 805         int                     status = -1;

 807         /*
 808          * setup the request data to attach to the libpcp msg
 809          */
 810         if (sc->req_ptr == NULL) {
 811                 if ((sc->req_ptr = (tsal_pcp_alarm_req_t *)kmem_zalloc(
 812                     sizeof (tsal_pcp_alarm_req_t),
 813                     KM_NOSLEEP)) == NULL)
 814                         goto alarm_return;
 815         }

 817         req_ptr = sc->req_ptr;

 819         if (new_state == ALARM_ON)
 820                 req_ptr->alarm_action = PCP_ALARM_ENABLE;
 821         else if (new_state == ALARM_OFF)
 822                 req_ptr->alarm_action = PCP_ALARM_DISABLE;

 824         req_ptr->alarm_id = alarm_type;

 826         send_msg.msg_type = PCP_ALARM_CONTROL;
 827         send_msg.sub_type = NULL;
 828         send_msg.msg_len = sizeof (tsal_pcp_alarm_req_t);
 829         send_msg.msg_data = (uint8_t *)req_ptr;

 831         /*
 832          * send the request, receive the response
 833          */
 834         if (tsal_pcp_send_recv(sc, &send_msg, &recv_msg,
 835             PCP_COMM_TIMEOUT) < 0) {
 836                 /* we either timed out or erred; either way try again */
 837                 (void) delay(drv_sectohz(PCP_COMM_TIMEOUT));
 837                 (void) delay(PCP_COMM_TIMEOUT * drv_usectohz(1000000));

 839                 if (tsal_pcp_send_recv(sc, &send_msg, &recv_msg,
 840                     PCP_COMM_TIMEOUT) < 0) {
 841                         goto alarm_return;
 842                 }
 843         }

 845         /*
 846          * validate that this data was meant for us
 847          */
 848         if (recv_msg.msg_type != PCP_ALARM_CONTROL_R) {
 849                 cmn_err(CE_WARN, "tsalarm: unbound packet received");
 850                 goto alarm_return;
 851         }
```

```
 853         /*
 854          * verify that the Alarm action has taken place
 855          */
 856         resp_ptr = (tsal_pcp_alarm_resp_t *)recv_msg.msg_data;
 857         if (resp_ptr->status == PCP_ALARM_ERROR) {
 858                 cmn_err(CE_WARN, "tsalarm: failed to set alarm status");
 859                 goto alarm_return;
 860         }

 862         /*
 863          * ensure the Alarm action taken is the one requested
 864          */
 865         if ((req_ptr->alarm_action == PCP_ALARM_DISABLE) &&
 866             (resp_ptr->alarm_state != ALARM_STATE_OFF)) {
 867                 cmn_err(CE_WARN, "tsalarm: failed to set alarm");
 868                 goto alarm_return;
 869         } else if ((req_ptr->alarm_action == PCP_ALARM_ENABLE) &&
 870             (resp_ptr->alarm_state != ALARM_STATE_ON)) {
 871                 cmn_err(CE_WARN, "tsalarm: failed to set alarm");
 872                 goto alarm_return;
 873         } else if (resp_ptr->alarm_state == ALARM_STATE_UNKNOWN) {
 874                 cmn_err(CE_WARN, "tsalarm: Alarm set to unknown state");
 875                 goto alarm_return;
 876         }

 878         status = TSAL_PCP_OK;

 880 alarm_return:
 881         return (status);
 882 }
```

_____***unchanged_portion_omitted_***

```
1049 /*
1050  * Function: wrapper for handling glvc calls (read/write/peek).
1051  */
1052 static int
1053 tsal_pcp_io_op(tsalarm_softc_t *sc, void *buf, int byte_cnt, int io_op)
1054 {
1055         int     rv;
1056         int     n;
1057         uint8_t *datap;
1058         int     (*func_ptr)(tsalarm_softc_t *, uint8_t *, int);
1059         int     io_sz;
1060         int     try_cnt;

1062         if ((buf == NULL) || (byte_cnt < 0)) {
1063                 return (TSAL_PCP_ERROR);
1064         }

1066         switch (io_op) {
1067                 case PCP_IO_OP_READ:
1068                         func_ptr = tsal_pcp_read;
1069                         break;
1070                 case PCP_IO_OP_WRITE:
1071                         func_ptr = tsal_pcp_write;
1072                         break;
1073                 case PCP_IO_OP_PEEK:
1074                         func_ptr = tsal_pcp_peek;
1075                         break;
1076                 default:
1077                         return (TSAL_PCP_ERROR);
1078         }

1080         /*
1081          * loop until all I/O done, try limit exceded, or real failure
1082          */
```



```
1084         rv = 0;
1085         datap = buf;
1086         while (rv < byte_cnt) {
1087                 io_sz = MIN((byte_cnt - rv), sc->mtu_size);
1088                 try_cnt = 0;
1089                 while ((n = (*func_ptr)(sc, datap, io_sz)) < 0) {
1090                         try_cnt++;
1091                         if (try_cnt > PCP_MAX_TRY_CNT) {
1092                                 rv = n;
1093                                 goto done;
1094                         }
1095                         /* waiting 5 secs. Do we need 5 Secs? */
1096                         (void) delay(drv_sectohz(PCP_GLVC_SLEEP));
1096                         (void) delay(PCP_GLVC_SLEEP * drv_usectohz(1000000));
1097                 } /* while trying the io operation */

1099                 if (n < 0) {
1100                         rv = n;
1101                         goto done;
1102                 }
1103                 rv += n;
1104                 datap += n;
1105         } /* while still have more data */

1107 done:
1108         if (rv == byte_cnt)
1109                 return (0);
1110         else
1111                 return (TSAL_PCP_ERROR);
1112 }
```

_____***unchanged_portion_omitted_***