```
**********************************************************
  444279 Mon Nov 10 10:43:57 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
5297 mptsas refhash replacement on reset can cause hang
**********************************************************
______unchanged_portion_omitted_
```

```
/*
 * Notes:
 *      Set up all device state and allocate data structures,
 *      mutexes, condition variables, etc. for device operation.
 *      Add interrupts needed.
 *      Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
 */
static int
mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        mptsas_t                *mpt = NULL;
        int                     instance, i, j;
        int                     doneq_thread_num;
        char                    intr_added = 0;
        char                    map_setup = 0;
        char                    config_setup = 0;
        char                    hba_attach_setup = 0;
        char                    smp_attach_setup = 0;
        char                    mutex_init_done = 0;
        char                    event_taskq_create = 0;
        char                    dr_taskq_create = 0;
        char                    doneq_thread_create = 0;
        char                    added_watchdog = 0;
        scsi_hba_tran_t         *hba_tran;
        uint_t                  mem_bar = MEM_SPACE;
        int                     rval = DDI_FAILURE;

        /* CONSTCOND */
        ASSERT(NO_COMPETING_THREADS);

        if (scsi_hba_iport_unit_address(dip)) {
                return (mptsas_iport_attach(dip, cmd));
        }

        switch (cmd) {
        case DDI_ATTACH:
                break;

        case DDI_RESUME:
                if ((hba_tran = ddi_get_driver_private(dip)) == NULL)
                        return (DDI_FAILURE);

                mpt = TRAN2MPT(hba_tran);

                if (!mpt) {
                        return (DDI_FAILURE);
                }

                /*
                 * Reset hardware and softc to "no outstanding commands"
                 * Note that a check condition can result on first command
                 * to a target.
                 */
                mutex_enter(&mpt->m_mutex);

                /*
                 * raise power.
                 */
                if (mpt->m_options & MPTSAS_OPT_PM) {
```

```
                        mutex_exit(&mpt->m_mutex);
                        (void) pm_busy_component(dip, 0);
                        rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
                        if (rval == DDI_SUCCESS) {
                                mutex_enter(&mpt->m_mutex);
                        } else {
                                /*
                                 * The pm_raise_power() call above failed,
                                 * and that can only occur if we were unable
                                 * to reset the hardware.  This is probably
                                 * due to unhealty hardware, and because
                                 * important filesystems(such as the root
                                 * filesystem) could be on the attached disks,
                                 * it would not be a good idea to continue,
                                 * as we won't be entirely certain we are
                                 * writing correct data.  So we panic() here
                                 * to not only prevent possible data corruption,
                                 * but to give developers or end users a hope
                                 * of identifying and correcting any problems.
                                 */
                                fm_panic("mptsas could not reset hardware "
                                    "during resume");
                        }
                }

                mpt->m_suspended = 0;

                /*
                 * Reinitialize ioc
                 */
                mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
                if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
                        mutex_exit(&mpt->m_mutex);
                        if (mpt->m_options & MPTSAS_OPT_PM) {
                                (void) pm_idle_component(dip, 0);
                        }
                        fm_panic("mptsas init chip fail during resume");
                }
                /*
                 * mptsas_update_driver_data needs interrupts so enable them
                 * first.
                 */
                MPTSAS_ENABLE_INTR(mpt);
                mptsas_update_driver_data(mpt);

                /* start requests, if possible */
                mptsas_restart_hba(mpt);

                mutex_exit(&mpt->m_mutex);

                /*
                 * Restart watch thread
                 */
                mutex_enter(&mptsas_global_mutex);
                if (mptsas_timeout_id == 0) {
                        mptsas_timeout_id = timeout(mptsas_watch, NULL,
                            mptsas_tick);
                        mptsas_timeouts_enabled = 1;
                }
                mutex_exit(&mptsas_global_mutex);

                /* report idle status to pm framework */
                if (mpt->m_options & MPTSAS_OPT_PM) {
                        (void) pm_idle_component(dip, 0);
                }
                return (DDI_SUCCESS);

        default:
                return (DDI_FAILURE);

        }

        instance = ddi_get_instance(dip);

        /*
         * Allocate softc information.
         */
        if (ddi_soft_state_zalloc(mptsas_state, instance) != DDI_SUCCESS) {
                mptsas_log(NULL, CE_WARN,
                    "mptsas%d: cannot allocate soft state", instance);
                goto fail;
        }

        mpt = ddi_get_soft_state(mptsas_state, instance);

        if (mpt == NULL) {
                mptsas_log(NULL, CE_WARN,
                    "mptsas%d: cannot get soft state", instance);
                goto fail;
        }

        /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
        scsi_size_clean(dip);

        mpt->m_dip = dip;
        mpt->m_instance = instance;

        /* Make a per-instance copy of the structures */
        mpt->m_io_dma_attr = mptsas_dma_attrs64;
        mpt->m_msg_dma_attr = mptsas_dma_attrs;
        mpt->m_reg_acc_attr = mptsas_dev_attr;
        mpt->m_dev_acc_attr = mptsas_dev_attr;

        /*
         * Initialize FMA
         */
        mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
            DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
            DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
            DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);

        mptsas_fm_init(mpt);

        if (mptsas_alloc_handshake_msg(mpt,
            sizeof (Mpi2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
                goto fail;
        }

        /*
         * Setup configuration space
         */
        if (mptsas_config_space_init(mpt) == FALSE) {
                mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
                goto fail;
        }
        config_setup++;

        if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
            0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "map setup failed");
```

```
                goto fail;
        }
        map_setup++;

        /*
         * A taskq is created for dealing with the event handler
         */
        if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
            1, TASKQ_DEFAULTPRI, 0)) == NULL) {
                mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
                goto fail;
        }
        event_taskq_create++;

        /*
         * A taskq is created for dealing with dr events
         */
        if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
            "mptsas_dr_taskq",
            1, TASKQ_DEFAULTPRI, 0)) == NULL) {
                mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
                    "failed");
                goto fail;
        }
        dr_taskq_create++;

        mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
            0, "mptsas_doneq_thread_threshold_prop", 10);
        mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
            0, "mptsas_doneq_length_threshold_prop", 8);
        mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
            0, "mptsas_doneq_thread_n_prop", 8);

        if (mpt->m_doneq_thread_n) {
                cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
                mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);

                mutex_enter(&mpt->m_doneq_mutex);
                mpt->m_doneq_thread_id =
                    kmem_zalloc(sizeof (mptsas_doneq_thread_list_t)
                    * mpt->m_doneq_thread_n, KM_SLEEP);

                for (j = 0; j < mpt->m_doneq_thread_n; j++) {
                        cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
                            CV_DRIVER, NULL);
                        mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
                            MUTEX_DRIVER, NULL);
                        mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
                        mpt->m_doneq_thread_id[j].flag |=
                            MPTSAS_DONEQ_THREAD_ACTIVE;
                        mpt->m_doneq_thread_id[j].arg.mpt = mpt;
                        mpt->m_doneq_thread_id[j].arg.t = j;
                        mpt->m_doneq_thread_id[j].threadp =
                            thread_create(NULL, 0, mptsas_doneq_thread,
                            &mpt->m_doneq_thread_id[j].arg,
                            0, &p0, TS_RUN, minclsyspri);
                        mpt->m_doneq_thread_id[j].donetail =
                            &mpt->m_doneq_thread_id[j].doneq;
                        mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
                }
                mutex_exit(&mpt->m_doneq_mutex);
                doneq_thread_create++;
        }

        /*
         * Disable hardware interrupt since we're not ready to
```

```
	 * handle it yet.
	 */
	MPTSAS_DISABLE_INTR(mpt);
	if (mptsas_register_intrs(mpt) == FALSE)
		goto fail;
	intr_added++;

	/* Initialize mutex used in interrupt handler */
	mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
	    DDI_INTR_PRI(mpt->m_intr_pri));
	mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
	    DDI_INTR_PRI(mpt->m_intr_pri));
	for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
		mutex_init(&mpt->m_phy_info[i].smhba_info.phy_mutex,
		    NULL, MUTEX_DRIVER,
		    DDI_INTR_PRI(mpt->m_intr_pri));
	}

	cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
	cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
	cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
	cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
	cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
	mutex_init_done++;

	mutex_enter(&mpt->m_mutex);
	/*
	 * Initialize power management component
	 */
	if (mpt->m_options & MPTSAS_OPT_PM) {
		if (mptsas_init_pm(mpt)) {
			mutex_exit(&mpt->m_mutex);
			mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
			    "failed");
			goto fail;
		}
	}

	/*
	 * Initialize chip using Message Unit Reset, if allowed
	 */
	mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
	if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
		mutex_exit(&mpt->m_mutex);
		mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
		goto fail;
	}

	mpt->m_targets = refhash_create(MPTSAS_TARGET_BUCKET_COUNT,
	    mptsas_target_addr_hash, mptsas_target_addr_cmp,
	    mptsas_target_free, sizeof (mptsas_target_t),
	    offsetof(mptsas_target_t, m_link),
	    offsetof(mptsas_target_t, m_addr), KM_SLEEP);

#endif /* ! codereview */
	/*
	 * Fill in the phy_info structure and get the base WWID
	 */
	if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
		mptsas_log(mpt, CE_WARN,
		    "mptsas_get_manufacture_page5 failed!");
		goto fail;
	}

	if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {
```

```
		mptsas_log(mpt, CE_WARN,
		    "mptsas_get_sas_io_unit_page_hndshk failed!");
		goto fail;
	}

	if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
		mptsas_log(mpt, CE_WARN,
		    "mptsas_get_manufacture_page0 failed!");
		goto fail;
	}

	mutex_exit(&mpt->m_mutex);

	/*
	 * Register the iport for multiple port HBA
	 */
	mptsas_iport_register(mpt);

	/*
	 * initialize SCSI HBA transport structure
	 */
	if (mptsas_hba_setup(mpt) == FALSE)
		goto fail;
	hba_attach_setup++;

	if (mptsas_smp_setup(mpt) == FALSE)
		goto fail;
	smp_attach_setup++;

	if (mptsas_cache_create(mpt) == FALSE)
		goto fail;

	mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
	    dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
	if (mpt->m_scsi_reset_delay == 0) {
		mptsas_log(mpt, CE_NOTE,
		    "scsi_reset_delay of 0 is not recommended,"
		    " resetting to SCSI_DEFAULT_RESET_DELAY\n");
		mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
	}

	/*
	 * Initialize the wait and done FIFO queue
	 */
	mpt->m_donetail = &mpt->m_doneq;
	mpt->m_waitqtail = &mpt->m_waitq;
	mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
	mpt->m_tx_draining = 0;

	/*
	 * ioc cmd queue initialize
	 */
	mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
	mpt->m_dev_handle = 0xFFFF;

	MPTSAS_ENABLE_INTR(mpt);

	/*
	 * enable event notification
	 */
	mutex_enter(&mpt->m_mutex);
	if (mptsas_ioc_enable_event_notification(mpt)) {
		mutex_exit(&mpt->m_mutex);
		goto fail;
	}
	mutex_exit(&mpt->m_mutex);
	/*
	 * used for mptsas_watch
	 */
	mptsas_list_add(mpt);

	mutex_enter(&mptsas_global_mutex);
	if (mptsas_timeouts_enabled == 0) {
		mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
		    dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);

		mptsas_tick = mptsas_scsi_watchdog_tick *
		    drv_usectohz((clock_t)1000000);

		mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
		mptsas_timeouts_enabled = 1;
	}
	mutex_exit(&mptsas_global_mutex);
	added_watchdog++;

	/*
	 * Initialize PHY info for smhba.
	 * This requires watchdog to be enabled otherwise if interrupts
	 * don't work the system will hang.
	 */
	if (mptsas_smhba_setup(mpt)) {
		mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
		    "failed");
		goto fail;
	}

	/* Check all dma handles allocated in attach */
	if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
	    != DDI_SUCCESS)) {
		goto fail;
	}

	/* Check all acc handles allocated in attach */
	if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
	    != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_config_handle)
	    != DDI_SUCCESS)) {
		goto fail;
	}

	/*
	 * After this point, we are not going to fail the attach.
	 */
```

```
	/* Print message of HBA present */
	ddi_report_dev(dip);

	/* report idle status to pm framework */
	if (mpt->m_options & MPTSAS_OPT_PM) {
		(void) pm_idle_component(dip, 0);
	}

	return (DDI_SUCCESS);

fail:
	mptsas_log(mpt, CE_WARN, "attach failed");
	mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
	ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
	if (mpt) {
		/* deallocate in reverse order */
		if (added_watchdog) {
			mptsas_list_del(mpt);
			mutex_enter(&mptsas_global_mutex);

			if (mptsas_timeout_id && (mptsas_head == NULL)) {
				timeout_id_t tid = mptsas_timeout_id;
				mptsas_timeouts_enabled = 0;
				mptsas_timeout_id = 0;
				mutex_exit(&mptsas_global_mutex);
				(void) untimeout(tid);
				mutex_enter(&mptsas_global_mutex);
			}
			mutex_exit(&mptsas_global_mutex);
		}

		mptsas_cache_destroy(mpt);

		if (smp_attach_setup) {
			mptsas_smp_teardown(mpt);
		}
		if (hba_attach_setup) {
			mptsas_hba_teardown(mpt);
		}

		if (mpt->m_targets)
			refhash_destroy(mpt->m_targets);
		if (mpt->m_smp_targets)
			refhash_destroy(mpt->m_smp_targets);

		if (mpt->m_active) {
			mptsas_free_active_slots(mpt);
		}
		if (intr_added) {
			mptsas_unregister_intrs(mpt);
		}

		if (doneq_thread_create) {
			mutex_enter(&mpt->m_doneq_mutex);
			doneq_thread_num = mpt->m_doneq_thread_n;
			for (j = 0; j < mpt->m_doneq_thread_n; j++) {
				mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
				mpt->m_doneq_thread_id[j].flag &=
				    (~MPTSAS_DONEQ_THREAD_ACTIVE);
				cv_signal(&mpt->m_doneq_thread_id[j].cv);
				mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
			}
			while (mpt->m_doneq_thread_n) {
				cv_wait(&mpt->m_doneq_thread_cv,
				    &mpt->m_doneq_mutex);
			}
			for (j = 0; j < doneq_thread_num; j++) {
				cv_destroy(&mpt->m_doneq_thread_id[j].cv);
				mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
			}
			kmem_free(mpt->m_doneq_thread_id,
			    sizeof (mptsas_doneq_thread_list_t)
			    * doneq_thread_num);
			mutex_exit(&mpt->m_doneq_mutex);
			cv_destroy(&mpt->m_doneq_thread_cv);
			mutex_destroy(&mpt->m_doneq_mutex);
		}
		if (event_taskq_create) {
			ddi_taskq_destroy(mpt->m_event_taskq);
		}
		if (dr_taskq_create) {
			ddi_taskq_destroy(mpt->m_dr_taskq);
		}
		if (mutex_init_done) {
			mutex_destroy(&mpt->m_tx_waitq_mutex);
			mutex_destroy(&mpt->m_passthru_mutex);
			mutex_destroy(&mpt->m_mutex);
			for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
				mutex_destroy(
				    &mpt->m_phy_info[i].smhba_info.phy_mutex);
			}
			cv_destroy(&mpt->m_cv);
			cv_destroy(&mpt->m_passthru_cv);
			cv_destroy(&mpt->m_fw_cv);
			cv_destroy(&mpt->m_config_cv);
			cv_destroy(&mpt->m_fw_diag_cv);
		}

		if (map_setup) {
			mptsas_cfg_fini(mpt);
		}
		if (config_setup) {
			mptsas_config_space_fini(mpt);
		}
		mptsas_free_handshake_msg(mpt);
		mptsas_hba_fini(mpt);

		mptsas_fm_fini(mpt);
		ddi_soft_state_free(mptsas_state, instance);
		ddi_prop_remove_all(dip);
	}
	return (DDI_FAILURE);
}

static int
mptsas_suspend(dev_info_t *devi)
{
	mptsas_t	*mpt, *g;
	scsi_hba_tran_t *tran;

	if (scsi_hba_iport_unit_address(devi)) {
		return (DDI_SUCCESS);
	}

	if ((tran = ddi_get_driver_private(devi)) == NULL)
		return (DDI_SUCCESS);

	mpt = TRAN2MPT(tran);
	if (!mpt) {
		return (DDI_SUCCESS);
	}
```

```
	mutex_enter(&mpt->m_mutex);

	if (mpt->m_suspended++) {
		mutex_exit(&mpt->m_mutex);
		return (DDI_SUCCESS);
	}

	/*
	 * Cancel timeout threads for this mpt
	 */
	if (mpt->m_quiesce_timeid) {
		timeout_id_t tid = mpt->m_quiesce_timeid;
		mpt->m_quiesce_timeid = 0;
		mutex_exit(&mpt->m_mutex);
		(void) untimeout(tid);
		mutex_enter(&mpt->m_mutex);
	}

	if (mpt->m_restart_cmd_timeid) {
		timeout_id_t tid = mpt->m_restart_cmd_timeid;
		mpt->m_restart_cmd_timeid = 0;
		mutex_exit(&mpt->m_mutex);
		(void) untimeout(tid);
		mutex_enter(&mpt->m_mutex);
	}

	mutex_exit(&mpt->m_mutex);

	(void) pm_idle_component(mpt->m_dip, 0);

	/*
	 * Cancel watch threads if all mpts suspended
	 */
	rw_enter(&mptsas_global_rwlock, RW_WRITER);
	for (g = mptsas_head; g != NULL; g = g->m_next) {
		if (!g->m_suspended)
			break;
	}
	rw_exit(&mptsas_global_rwlock);

	mutex_enter(&mptsas_global_mutex);
	if (g == NULL) {
		timeout_id_t tid;

		mptsas_timeouts_enabled = 0;
		if (mptsas_timeout_id) {
			tid = mptsas_timeout_id;
			mptsas_timeout_id = 0;
			mutex_exit(&mptsas_global_mutex);
			(void) untimeout(tid);
			mutex_enter(&mptsas_global_mutex);
		}
		if (mptsas_reset_watch) {
			tid = mptsas_reset_watch;
			mptsas_reset_watch = 0;
			mutex_exit(&mptsas_global_mutex);
			(void) untimeout(tid);
			mutex_enter(&mptsas_global_mutex);
		}
	}
	mutex_exit(&mptsas_global_mutex);

	mutex_enter(&mpt->m_mutex);

	/*
	 * If this mpt is not in full power(PM_LEVEL_D0), just return.
```

```
         */
        if ((mpt->m_options & MPTSAS_OPT_PM) &&
            (mpt->m_power_level != PM_LEVEL_D0)) {
                mutex_exit(&mpt->m_mutex);
                return (DDI_SUCCESS);
        }

        /* Disable HBA interrupts in hardware */
        MPTSAS_DISABLE_INTR(mpt);
        /*
         * Send RAID action system shutdown to sync IR
         */
        mptsas_raid_action_system_shutdown(mpt);

        mutex_exit(&mpt->m_mutex);

        /* drain the taskq */
        ddi_taskq_wait(mpt->m_event_taskq);
        ddi_taskq_wait(mpt->m_dr_taskq);

        return (DDI_SUCCESS);
}

#ifdef  __sparc
/*ARGSUSED*/
static int
mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd)
{
        mptsas_t        *mpt;
        scsi_hba_tran_t *tran;

        /*
         * If this call is for iport, just return.
         */
        if (scsi_hba_iport_unit_address(devi))
                return (DDI_SUCCESS);

        if ((tran = ddi_get_driver_private(devi)) == NULL)
                return (DDI_SUCCESS);

        if ((mpt = TRAN2MPT(tran)) == NULL)
                return (DDI_SUCCESS);

        /*
         * Send RAID action system shutdown to sync IR.  Disable HBA
         * interrupts in hardware first.
         */
        MPTSAS_DISABLE_INTR(mpt);
        mptsas_raid_action_system_shutdown(mpt);

        return (DDI_SUCCESS);
}
#else /* __sparc */
/*
 * quiesce(9E) entry point.
 *
 * This function is called when the system is single-threaded at high
 * PIL with preemption disabled. Therefore, this function must not be
 * blocked.
 *
 * This function returns DDI_SUCCESS on success, or DDI_FAILURE on failure.
 * DDI_FAILURE indicates an error condition and should almost never happen.
 */
static int
mptsas_quiesce(dev_info_t *devi)
{
```

```
        mptsas_t        *mpt;
        scsi_hba_tran_t *tran;

        /*
         * If this call is for iport, just return.
         */
        if (scsi_hba_iport_unit_address(devi))
                return (DDI_SUCCESS);

        if ((tran = ddi_get_driver_private(devi)) == NULL)
                return (DDI_SUCCESS);

        if ((mpt = TRAN2MPT(tran)) == NULL)
                return (DDI_SUCCESS);

        /* Disable HBA interrupts in hardware */
        MPTSAS_DISABLE_INTR(mpt);
        /* Send RAID action system shutdonw to sync IR */
        mptsas_raid_action_system_shutdown(mpt);

        return (DDI_SUCCESS);
}
#endif  /* __sparc */

/*
 * detach(9E).  Remove all device allocations and system resources;
 * disable device interrupts.
 * Return DDI_SUCCESS if done; DDI_FAILURE if there’s a problem.
 */
static int
mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
{
        /* CONSTCOND */
        ASSERT(NO_COMPETING_THREADS);
        NDBG0(("mptsas_detach: dip=0x%p cmd=0x%p", (void *)devi, (void *)cmd));

        switch (cmd) {
        case DDI_DETACH:
                return (mptsas_do_detach(devi));

        case DDI_SUSPEND:
                return (mptsas_suspend(devi));

        default:
                return (DDI_FAILURE);
        }
        /* NOTREACHED */
}

static int
mptsas_do_detach(dev_info_t *dip)
{
        mptsas_t        *mpt;
        scsi_hba_tran_t *tran;
        int             circ = 0;
        int             circ1 = 0;
        mdi_pathinfo_t  *pip = NULL;
        int             i;
        int             doneq_thread_num = 0;

        NDBG0(("mptsas_do_detach: dip=0x%p", (void *)dip));

        if ((tran = ndi_flavorv_get(dip, SCSA_FLAVOR_SCSI_DEVICE)) == NULL)
                return (DDI_FAILURE);

        mpt = TRAN2MPT(tran);
	if (!mpt) {
		return (DDI_FAILURE);
	}
	/*
	 * Still have pathinfo child, should not detach mpt driver
	 */
	if (scsi_hba_iport_unit_address(dip)) {
		if (mpt->m_mpxio_enable) {
			/*
			 * MPxIO enabled for the iport
			 */
			ndi_devi_enter(scsi_vhci_dip, &circ1);
			ndi_devi_enter(dip, &circ);
			while (pip = mdi_get_next_client_path(dip, NULL)) {
				if (mdi_pi_free(pip, 0) == MDI_SUCCESS) {
					continue;
				}
				ndi_devi_exit(dip, circ);
				ndi_devi_exit(scsi_vhci_dip, circ1);
				NDBG12(("detach failed because of "
				    "outstanding path info"));
				return (DDI_FAILURE);
			}
			ndi_devi_exit(dip, circ);
			ndi_devi_exit(scsi_vhci_dip, circ1);
			(void) mdi_phci_unregister(dip, 0);
		}

		ddi_prop_remove_all(dip);

		return (DDI_SUCCESS);
	}

	/* Make sure power level is D0 before accessing registers */
	if (mpt->m_options & MPTSAS_OPT_PM) {
		(void) pm_busy_component(dip, 0);
		if (mpt->m_power_level != PM_LEVEL_D0) {
			if (pm_raise_power(dip, 0, PM_LEVEL_D0) !=
			    DDI_SUCCESS) {
				mptsas_log(mpt, CE_WARN,
				    "mptsas%d: Raise power request failed.",
				    mpt->m_instance);
				(void) pm_idle_component(dip, 0);
				return (DDI_FAILURE);
			}
		}
	}

	/*
	 * Send RAID action system shutdown to sync IR.  After action, send a
	 * Message Unit Reset. Since after that DMA resource will be freed,
	 * set ioc to READY state will avoid HBA initiated DMA operation.
	 */
	mutex_enter(&mpt->m_mutex);
	MPTSAS_DISABLE_INTR(mpt);
	mptsas_raid_action_system_shutdown(mpt);
	mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
	(void) mptsas_ioc_reset(mpt, FALSE);
	mutex_exit(&mpt->m_mutex);
	mptsas_rem_intrs(mpt);
	ddi_taskq_destroy(mpt->m_event_taskq);
	ddi_taskq_destroy(mpt->m_dr_taskq);

	if (mpt->m_doneq_thread_n) {
		mutex_enter(&mpt->m_doneq_mutex);
		doneq_thread_num = mpt->m_doneq_thread_n;
```

```
		for (i = 0; i < mpt->m_doneq_thread_n; i++) {
			mutex_enter(&mpt->m_doneq_thread_id[i].mutex);
			mpt->m_doneq_thread_id[i].flag &=
			    (~MPTSAS_DONEQ_THREAD_ACTIVE);
			cv_signal(&mpt->m_doneq_thread_id[i].cv);
			mutex_exit(&mpt->m_doneq_thread_id[i].mutex);
		}
		while (mpt->m_doneq_thread_n) {
			cv_wait(&mpt->m_doneq_thread_cv,
			    &mpt->m_doneq_mutex);
		}
		for (i = 0;  i < doneq_thread_num; i++) {
			cv_destroy(&mpt->m_doneq_thread_id[i].cv);
			mutex_destroy(&mpt->m_doneq_thread_id[i].mutex);
		}
		kmem_free(mpt->m_doneq_thread_id,
		    sizeof (mptsas_doneq_thread_list_t)
		    * doneq_thread_num);
		mutex_exit(&mpt->m_doneq_mutex);
		cv_destroy(&mpt->m_doneq_thread_cv);
		mutex_destroy(&mpt->m_doneq_mutex);
	}

	scsi_hba_reset_notify_tear_down(mpt->m_reset_notify_listf);

	mptsas_list_del(mpt);

	/*
	 * Cancel timeout threads for this mpt
	 */
	mutex_enter(&mpt->m_mutex);
	if (mpt->m_quiesce_timeid) {
		timeout_id_t tid = mpt->m_quiesce_timeid;
		mpt->m_quiesce_timeid = 0;
		mutex_exit(&mpt->m_mutex);
		(void) untimeout(tid);
		mutex_enter(&mpt->m_mutex);
	}

	if (mpt->m_restart_cmd_timeid) {
		timeout_id_t tid = mpt->m_restart_cmd_timeid;
		mpt->m_restart_cmd_timeid = 0;
		mutex_exit(&mpt->m_mutex);
		(void) untimeout(tid);
		mutex_enter(&mpt->m_mutex);
	}

	mutex_exit(&mpt->m_mutex);

	/*
	 * last mpt? ... if active, CANCEL watch threads.
	 */
	mutex_enter(&mptsas_global_mutex);
	if (mptsas_head == NULL) {
		timeout_id_t tid;
		/*
		 * Clear mptsas_timeouts_enable so that the watch thread
		 * gets restarted on DDI_ATTACH
		 */
		mptsas_timeouts_enabled = 0;
		if (mptsas_timeout_id) {
			tid = mptsas_timeout_id;
			mptsas_timeout_id = 0;
			mutex_exit(&mptsas_global_mutex);
			(void) untimeout(tid);
			mutex_enter(&mptsas_global_mutex);
		}
		if (mptsas_reset_watch) {
			tid = mptsas_reset_watch;
			mptsas_reset_watch = 0;
			mutex_exit(&mptsas_global_mutex);
			(void) untimeout(tid);
			mutex_enter(&mptsas_global_mutex);
		}
	}
	mutex_exit(&mptsas_global_mutex);

	/*
	 * Delete Phy stats
	 */
	mptsas_destroy_phy_stats(mpt);

	mptsas_destroy_hashes(mpt);

	/*
	 * Delete nt_active.
	 */
	mutex_enter(&mpt->m_mutex);
	mptsas_free_active_slots(mpt);
	mutex_exit(&mpt->m_mutex);

	/* deallocate everything that was allocated in mptsas_attach */
	mptsas_cache_destroy(mpt);

	mptsas_hba_fini(mpt);
	mptsas_cfg_fini(mpt);

	/* Lower the power informing PM Framework */
	if (mpt->m_options & MPTSAS_OPT_PM) {
		if (pm_lower_power(dip, 0, PM_LEVEL_D3) != DDI_SUCCESS)
			mptsas_log(mpt, CE_WARN,
			    "!mptsas%d: Lower power request failed "
			    "during detach, ignoring.",
			    mpt->m_instance);
	}

	mutex_destroy(&mpt->m_tx_waitq_mutex);
	mutex_destroy(&mpt->m_passthru_mutex);
	mutex_destroy(&mpt->m_mutex);
	for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
		mutex_destroy(&mpt->m_phy_info[i].smhba_info.phy_mutex);
	}
	cv_destroy(&mpt->m_cv);
	cv_destroy(&mpt->m_passthru_cv);
	cv_destroy(&mpt->m_fw_cv);
	cv_destroy(&mpt->m_config_cv);
	cv_destroy(&mpt->m_fw_diag_cv);


	mptsas_smp_teardown(mpt);
	mptsas_hba_teardown(mpt);

	mptsas_config_space_fini(mpt);

	mptsas_free_handshake_msg(mpt);

	mptsas_fm_fini(mpt);
	ddi_soft_state_free(mptsas_state, ddi_get_instance(dip));
	ddi_prop_remove_all(dip);

	return (DDI_SUCCESS);
}
```

```
static void
mptsas_list_add(mptsas_t *mpt)
{
	rw_enter(&mptsas_global_rwlock, RW_WRITER);

	if (mptsas_head == NULL) {
		mptsas_head = mpt;
	} else {
		mptsas_tail->m_next = mpt;
	}
	mptsas_tail = mpt;
	rw_exit(&mptsas_global_rwlock);
}

static void
mptsas_list_del(mptsas_t *mpt)
{
	mptsas_t *m;
	/*
	 * Remove device instance from the global linked list
	 */
	rw_enter(&mptsas_global_rwlock, RW_WRITER);
	if (mptsas_head == mpt) {
		m = mptsas_head = mpt->m_next;
	} else {
		for (m = mptsas_head; m != NULL; m = m->m_next) {
			if (m->m_next == mpt) {
				m->m_next = mpt->m_next;
				break;
			}
		}
		if (m == NULL) {
			mptsas_log(mpt, CE_PANIC, "Not in softc list!");
		}
	}

	if (mptsas_tail == mpt) {
		mptsas_tail = m;
	}
	rw_exit(&mptsas_global_rwlock);
}

static int
mptsas_alloc_handshake_msg(mptsas_t *mpt, size_t alloc_size)
{
	ddi_dma_attr_t  task_dma_attrs;

	mpt->m_hshk_dma_size = 0;
	task_dma_attrs = mpt->m_msg_dma_attr;
	task_dma_attrs.dma_attr_sgllen = 1;
	task_dma_attrs.dma_attr_granular = (uint32_t)(alloc_size);

	/* allocate Task Management ddi_dma resources */
	if (mptsas_dma_addr_create(mpt, task_dma_attrs,
	    &mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl, &mpt->m_hshk_memp,
	    alloc_size, NULL) == FALSE) {
		return (DDI_FAILURE);
	}
	mpt->m_hshk_dma_size = alloc_size;

	return (DDI_SUCCESS);
}

static void
mptsas_free_handshake_msg(mptsas_t *mpt)
```

```
{
        if (mpt->m_hshk_dma_size == 0)
                return;
        mptsas_dma_addr_destroy(&mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl);
        mpt->m_hshk_dma_size = 0;
}

static int
mptsas_hba_setup(mptsas_t *mpt)
{
        scsi_hba_tran_t         *hba_tran;
        int                     tran_flags;

        /* Allocate a transport structure */
        hba_tran = mpt->m_tran = scsi_hba_tran_alloc(mpt->m_dip,
            SCSI_HBA_CANSLEEP);
        ASSERT(mpt->m_tran != NULL);

        hba_tran->tran_hba_private      = mpt;
        hba_tran->tran_tgt_private      = NULL;

        hba_tran->tran_tgt_init         = mptsas_scsi_tgt_init;
        hba_tran->tran_tgt_free         = mptsas_scsi_tgt_free;

        hba_tran->tran_start            = mptsas_scsi_start;
        hba_tran->tran_reset            = mptsas_scsi_reset;
        hba_tran->tran_abort            = mptsas_scsi_abort;
        hba_tran->tran_getcap           = mptsas_scsi_getcap;
        hba_tran->tran_setcap           = mptsas_scsi_setcap;
        hba_tran->tran_init_pkt         = mptsas_scsi_init_pkt;
        hba_tran->tran_destroy_pkt      = mptsas_scsi_destroy_pkt;

        hba_tran->tran_dmafree          = mptsas_scsi_dmafree;
        hba_tran->tran_sync_pkt         = mptsas_scsi_sync_pkt;
        hba_tran->tran_reset_notify     = mptsas_scsi_reset_notify;

        hba_tran->tran_get_bus_addr     = mptsas_get_bus_addr;
        hba_tran->tran_get_name         = mptsas_get_name;

        hba_tran->tran_quiesce          = mptsas_scsi_quiesce;
        hba_tran->tran_unquiesce        = mptsas_scsi_unquiesce;
        hba_tran->tran_bus_reset        = NULL;

        hba_tran->tran_add_eventcall    = NULL;
        hba_tran->tran_get_eventcookie  = NULL;
        hba_tran->tran_post_event       = NULL;
        hba_tran->tran_remove_eventcall = NULL;

        hba_tran->tran_bus_config       = mptsas_bus_config;

        hba_tran->tran_interconnect_type = INTERCONNECT_SAS;

        /*
         * All children of the HBA are iports. We need tran was cloned.
         * So we pass the flags to SCSA. SCSI_HBA_TRAN_CLONE will be
         * inherited to iport’s tran vector.
         */
        tran_flags = (SCSI_HBA_HBA | SCSI_HBA_TRAN_CLONE);

        if (scsi_hba_attach_setup(mpt->m_dip, &mpt->m_msg_dma_attr,
            hba_tran, tran_flags) != DDI_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "hba attach setup failed");
                scsi_hba_tran_free(hba_tran);
                mpt->m_tran = NULL;
                return (FALSE);
        }
```

```
        return (TRUE);
}

static void
mptsas_hba_teardown(mptsas_t *mpt)
{
        (void) scsi_hba_detach(mpt->m_dip);
        if (mpt->m_tran != NULL) {
                scsi_hba_tran_free(mpt->m_tran);
                mpt->m_tran = NULL;
        }
}

static void
mptsas_iport_register(mptsas_t *mpt)
{
        int i, j;
        mptsas_phymask_t        mask = 0x0;
        /*
         * initial value of mask is 0
         */
        mutex_enter(&mpt->m_mutex);
        for (i = 0; i < mpt->m_num_phys; i++) {
                mptsas_phymask_t phy_mask = 0x0;
                char phy_mask_name[MPTSAS_MAX_PHYS];
                uint8_t current_port;

                if (mpt->m_phy_info[i].attached_devhdl == 0)
                        continue;

                bzero(phy_mask_name, sizeof (phy_mask_name));

                current_port = mpt->m_phy_info[i].port_num;

                if ((mask & (1 << i)) != 0)
                        continue;

                for (j = 0; j < mpt->m_num_phys; j++) {
                        if (mpt->m_phy_info[j].attached_devhdl &&
                            (mpt->m_phy_info[j].port_num == current_port)) {
                                phy_mask |= (1 << j);
                        }
                }
                mask = mask | phy_mask;

                for (j = 0; j < mpt->m_num_phys; j++) {
                        if ((phy_mask >> j) & 0x01) {
                                mpt->m_phy_info[j].phy_mask = phy_mask;
                        }
                }

                (void) sprintf(phy_mask_name, "%x", phy_mask);

                mutex_exit(&mpt->m_mutex);
                /*
                 * register a iport
                 */
                (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
                mutex_enter(&mpt->m_mutex);
        }
        mutex_exit(&mpt->m_mutex);
        /*
         * register a virtual port for RAID volume always
         */
        (void) scsi_hba_iport_register(mpt->m_dip, "v0");
```

```
}

static int
mptsas_smp_setup(mptsas_t *mpt)
{
        mpt->m_smptran = smp_hba_tran_alloc(mpt->m_dip);
        ASSERT(mpt->m_smptran != NULL);
        mpt->m_smptran->smp_tran_hba_private = mpt;
        mpt->m_smptran->smp_tran_start = mptsas_smp_start;
        if (smp_hba_attach_setup(mpt->m_dip, mpt->m_smptran) != DDI_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "smp attach setup failed");
                smp_hba_tran_free(mpt->m_smptran);
                mpt->m_smptran = NULL;
                return (FALSE);
        }
        /*
         * Initialize smp hash table
         */
        mpt->m_smp_targets = refhash_create(MPTSAS_SMP_BUCKET_COUNT,
            mptsas_target_addr_hash, mptsas_target_addr_cmp,
            mptsas_smp_free, sizeof (mptsas_smp_t),
            offsetof(mptsas_smp_t, m_link), offsetof(mptsas_smp_t, m_addr),
            KM_SLEEP);
        mpt->m_smp_devhdl = 0xFFFF;

        return (TRUE);
}

static void
mptsas_smp_teardown(mptsas_t *mpt)
{
        (void) smp_hba_detach(mpt->m_dip);
        if (mpt->m_smptran != NULL) {
                smp_hba_tran_free(mpt->m_smptran);
                mpt->m_smptran = NULL;
        }
        mpt->m_smp_devhdl = 0;
}

static int
mptsas_cache_create(mptsas_t *mpt)
{
        int instance = mpt->m_instance;
        char buf[64];

        /*
         * create kmem cache for packets
         */
        (void) sprintf(buf, "mptsas%d_cache", instance);
        mpt->m_kmem_cache = kmem_cache_create(buf,
            sizeof (struct mptsas_cmd) + scsi_pkt_size(), 8,
            mptsas_kmem_cache_constructor, mptsas_kmem_cache_destructor,
            NULL, (void *)mpt, NULL, 0);

        if (mpt->m_kmem_cache == NULL) {
                mptsas_log(mpt, CE_WARN, "creating kmem cache failed");
                return (FALSE);
        }

        /*
         * create kmem cache for extra SGL frames if SGL cannot
         * be accomodated into main request frame.
         */
        (void) sprintf(buf, "mptsas%d_cache_frames", instance);
        mpt->m_cache_frames = kmem_cache_create(buf,
            sizeof (mptsas_cache_frames_t), 8,
```

```
            mptsas_cache_frames_constructor, mptsas_cache_frames_destructor,
            NULL, (void *)mpt, NULL, 0);

        if (mpt->m_cache_frames == NULL) {
                mptsas_log(mpt, CE_WARN, "creating cache for frames failed");
                return (FALSE);
        }

        return (TRUE);
}

static void
mptsas_cache_destroy(mptsas_t *mpt)
{
        /* deallocate in reverse order */
        if (mpt->m_cache_frames) {
                kmem_cache_destroy(mpt->m_cache_frames);
                mpt->m_cache_frames = NULL;
        }
        if (mpt->m_kmem_cache) {
                kmem_cache_destroy(mpt->m_kmem_cache);
                mpt->m_kmem_cache = NULL;
        }
}

static int
mptsas_power(dev_info_t *dip, int component, int level)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(component))
#endif
        mptsas_t        *mpt;
        int             rval = DDI_SUCCESS;
        int             polls = 0;
        uint32_t        ioc_status;

        if (scsi_hba_iport_unit_address(dip) != 0)
                return (DDI_SUCCESS);

        mpt = ddi_get_soft_state(mptsas_state, ddi_get_instance(dip));
        if (mpt == NULL) {
                return (DDI_FAILURE);
        }

        mutex_enter(&mpt->m_mutex);

        /*
         * If the device is busy, don't lower its power level
         */
        if (mpt->m_busy && (mpt->m_power_level > level)) {
                mutex_exit(&mpt->m_mutex);
                return (DDI_FAILURE);
        }
        switch (level) {
        case PM_LEVEL_D0:
                NDBG11(("mptsas%d: turning power ON.", mpt->m_instance));
                MPTSAS_POWER_ON(mpt);
                /*
                 * Wait up to 30 seconds for IOC to come out of reset.
                 */
                while (((ioc_status = ddi_get32(mpt->m_datap,
                    &mpt->m_reg->Doorbell)) &
                    MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_RESET) {
                        if (polls++ > 3000) {
                                break;
                        }
```

```
                                delay(drv_usectohz(10000));
                        }
                        /*
                         * If IOC is not in operational state, try to hard reset it.
                         */
                        if ((ioc_status & MPI2_IOC_STATE_MASK) !=
                            MPI2_IOC_STATE_OPERATIONAL) {
                                mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
                                if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
                                        mptsas_log(mpt, CE_WARN,
                                            "mptsas_power: hard reset failed");
                                        mutex_exit(&mpt->m_mutex);
                                        return (DDI_FAILURE);
                                }
                        }
                        mpt->m_power_level = PM_LEVEL_D0;
                        break;
        case PM_LEVEL_D3:
                NDBG11(("mptsas%d: turning power OFF.", mpt->m_instance));
                MPTSAS_POWER_OFF(mpt);
                break;
        default:
                mptsas_log(mpt, CE_WARN, "mptsas%d: unknown power level <%x>.",
                    mpt->m_instance, level);
                rval = DDI_FAILURE;
                break;
        }
        mutex_exit(&mpt->m_mutex);
        return (rval);
}

/*
 * Initialize configuration space and figure out which
 * chip and revison of the chip the mpt driver is using.
 */
static int
mptsas_config_space_init(mptsas_t *mpt)
{
        NDBG0(("mptsas_config_space_init"));

        if (mpt->m_config_handle != NULL)
                return (TRUE);

        if (pci_config_setup(mpt->m_dip,
            &mpt->m_config_handle) != DDI_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "cannot map configuration space.");
                return (FALSE);
        }

        /*
         * This is a workaround for a XMITS ASIC bug which does not
         * drive the CBE upper bits.
         */
        if (pci_config_get16(mpt->m_config_handle, PCI_CONF_STAT) &
            PCI_STAT_PERROR) {
                pci_config_put16(mpt->m_config_handle, PCI_CONF_STAT,
                    PCI_STAT_PERROR);
        }

        mptsas_setup_cmd_reg(mpt);

        /*
         * Get the chip device id:
         */
        mpt->m_devid = pci_config_get16(mpt->m_config_handle, PCI_CONF_DEVID);
```

```
        /*
         * Save the revision.
         */
        mpt->m_revid = pci_config_get8(mpt->m_config_handle, PCI_CONF_REVID);

        /*
         * Save the SubSystem Vendor and Device IDs
         */
        mpt->m_svid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBVENID);
        mpt->m_ssid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBSYSID);

        /*
         * Set the latency timer to 0x40 as specified by the upa -> pci
         * bridge chip design team.  This may be done by the sparc pci
         * bus nexus driver, but the driver should make sure the latency
         * timer is correct for performance reasons.
         */
        pci_config_put8(mpt->m_config_handle, PCI_CONF_LATENCY_TIMER,
            MPTSAS_LATENCY_TIMER);

        (void) mptsas_get_pci_cap(mpt);
        return (TRUE);
}

static void
mptsas_config_space_fini(mptsas_t *mpt)
{
        if (mpt->m_config_handle != NULL) {
                mptsas_disable_bus_master(mpt);
                pci_config_teardown(&mpt->m_config_handle);
                mpt->m_config_handle = NULL;
        }
}

static void
mptsas_setup_cmd_reg(mptsas_t *mpt)
{
        ushort_t        cmdreg;

        /*
         * Set the command register to the needed values.
         */
        cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
        cmdreg |= (PCI_COMM_ME | PCI_COMM_SERR_ENABLE |
            PCI_COMM_PARITY_DETECT | PCI_COMM_MAE);
        cmdreg &= ~PCI_COMM_IO;
        pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
}

static void
mptsas_disable_bus_master(mptsas_t *mpt)
{
        ushort_t        cmdreg;

        /*
         * Clear the master enable bit in the PCI command register.
         * This prevents any bus mastering activity like DMA.
         */
        cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
        cmdreg &= ~PCI_COMM_ME;
        pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
}

int
mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_statep)
{
        ddi_dma_attr_t  attrs;

        attrs = mpt->m_io_dma_attr;
        attrs.dma_attr_sgllen = 1;

        ASSERT(dma_statep != NULL);

        if (mptsas_dma_addr_create(mpt, attrs, &dma_statep->handle,
            &dma_statep->accessp, &dma_statep->memp, dma_statep->size,
            &dma_statep->cookie) == FALSE) {
                return (DDI_FAILURE);
        }

        return (DDI_SUCCESS);
}

void
mptsas_dma_free(mptsas_dma_alloc_state_t *dma_statep)
{
        ASSERT(dma_statep != NULL);
        mptsas_dma_addr_destroy(&dma_statep->handle, &dma_statep->accessp);
        dma_statep->size = 0;
}

int
mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)())
{
        ddi_dma_attr_t          attrs;
        ddi_dma_handle_t        dma_handle;
        caddr_t                 memp;
        ddi_acc_handle_t        accessp;
        int                     rval;

        ASSERT(mutex_owned(&mpt->m_mutex));

        attrs = mpt->m_msg_dma_attr;
        attrs.dma_attr_sgllen = 1;
        attrs.dma_attr_granular = size;

        if (mptsas_dma_addr_create(mpt, attrs, &dma_handle,
            &accessp, &memp, size, NULL) == FALSE) {
                return (DDI_FAILURE);
        }

        rval = (*callback) (mpt, memp, var, accessp);

        if ((mptsas_check_dma_handle(dma_handle) != DDI_SUCCESS) ||
            (mptsas_check_acc_handle(accessp) != DDI_SUCCESS)) {
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
                rval = DDI_FAILURE;
        }

        mptsas_dma_addr_destroy(&dma_handle, &accessp);
        return (rval);

}

static int
mptsas_alloc_request_frames(mptsas_t *mpt)
{
        ddi_dma_attr_t          frame_dma_attrs;
        caddr_t                 memp;
        ddi_dma_cookie_t        cookie;
        size_t                  mem_size;

        /*
```

```
         * re-alloc when it has already alloced
         */
        mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
            &mpt->m_acc_req_frame_hdl);

        /*
         * The size of the request frame pool is:
         *   Number of Request Frames * Request Frame Size
         */
        mem_size = mpt->m_max_requests * mpt->m_req_frame_size;

        /*
         * set the DMA attributes.  System Request Message Frames must be
         * aligned on a 16-byte boundry.
         */
        frame_dma_attrs = mpt->m_msg_dma_attr;
        frame_dma_attrs.dma_attr_align = 16;
        frame_dma_attrs.dma_attr_sgllen = 1;

        /*
         * allocate the request frame pool.
         */
        if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
            &mpt->m_dma_req_frame_hdl, &mpt->m_acc_req_frame_hdl, &memp,
            mem_size, &cookie) == FALSE) {
                return (DDI_FAILURE);
        }

        /*
         * Store the request frame memory address.  This chip uses this
         * address to dma to and from the driver's frame.  The second
         * address is the address mpt uses to fill in the frame.
         */
        mpt->m_req_frame_dma_addr = cookie.dmac_laddress;
        mpt->m_req_frame = memp;

        /*
         * Clear the request frame pool.
         */
        bzero(mpt->m_req_frame, mem_size);

        return (DDI_SUCCESS);
}

static int
mptsas_alloc_reply_frames(mptsas_t *mpt)
{
        ddi_dma_attr_t          frame_dma_attrs;
        caddr_t                 memp;
        ddi_dma_cookie_t        cookie;
        size_t                  mem_size;

        /*
         * re-alloc when it has already alloced
         */
        mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
            &mpt->m_acc_reply_frame_hdl);

        /*
         * The size of the reply frame pool is:
         *   Number of Reply Frames * Reply Frame Size
         */
        mem_size = mpt->m_max_replies * mpt->m_reply_frame_size;

        /*
         * set the DMA attributes.   System Reply Message Frames must be
         * aligned on a 4-byte boundry.  This is the default.
         */
        frame_dma_attrs = mpt->m_msg_dma_attr;
        frame_dma_attrs.dma_attr_sgllen = 1;

        /*
         * allocate the reply frame pool
         */
        if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
            &mpt->m_dma_reply_frame_hdl, &mpt->m_acc_reply_frame_hdl, &memp,
            mem_size, &cookie) == FALSE) {
                return (DDI_FAILURE);
        }

        /*
         * Store the reply frame memory address.  This chip uses this
         * address to dma to and from the driver's frame.  The second
         * address is the address mpt uses to process the frame.
         */
        mpt->m_reply_frame_dma_addr = cookie.dmac_laddress;
        mpt->m_reply_frame = memp;

        /*
         * Clear the reply frame pool.
         */
        bzero(mpt->m_reply_frame, mem_size);

        return (DDI_SUCCESS);
}

static int
mptsas_alloc_free_queue(mptsas_t *mpt)
{
        ddi_dma_attr_t          frame_dma_attrs;
        caddr_t                 memp;
        ddi_dma_cookie_t        cookie;
        size_t                  mem_size;

        /*
         * re-alloc when it has already alloced
         */
        mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
            &mpt->m_acc_free_queue_hdl);

        /*
         * The reply free queue size is:
         *   Reply Free Queue Depth * 4
         * The "4" is the size of one 32 bit address (low part of 64-bit
         *   address)
         */
        mem_size = mpt->m_free_queue_depth * 4;

        /*
         * set the DMA attributes  The Reply Free Queue must be aligned on a
         * 16-byte boundry.
         */
        frame_dma_attrs = mpt->m_msg_dma_attr;
        frame_dma_attrs.dma_attr_align = 16;
        frame_dma_attrs.dma_attr_sgllen = 1;

        /*
         * allocate the reply free queue
         */
        if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
            &mpt->m_dma_free_queue_hdl, &mpt->m_acc_free_queue_hdl, &memp,
            mem_size, &cookie) == FALSE) {
```

```
                return (DDI_FAILURE);
        }

        /*
         * Store the reply free queue memory address.  This chip uses this
         * address to read from the reply free queue.  The second address
         * is the address mpt uses to manage the queue.
         */
        mpt->m_free_queue_dma_addr = cookie.dmac_laddress;
        mpt->m_free_queue = memp;

        /*
         * Clear the reply free queue memory.
         */
        bzero(mpt->m_free_queue, mem_size);

        return (DDI_SUCCESS);
}

static int
mptsas_alloc_post_queue(mptsas_t *mpt)
{
        ddi_dma_attr_t          frame_dma_attrs;
        caddr_t                 memp;
        ddi_dma_cookie_t        cookie;
        size_t                  mem_size;

        /*
         * re-alloc when it has already alloced
         */
        mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
            &mpt->m_acc_post_queue_hdl);

        /*
         * The reply descriptor post queue size is:
         *   Reply Descriptor Post Queue Depth * 8
         * The "8" is the size of each descriptor (8 bytes or 64 bits).
         */
        mem_size = mpt->m_post_queue_depth * 8;

        /*
         * set the DMA attributes.  The Reply Descriptor Post Queue must be
         * aligned on a 16-byte boundry.
         */
        frame_dma_attrs = mpt->m_msg_dma_attr;
        frame_dma_attrs.dma_attr_align = 16;
        frame_dma_attrs.dma_attr_sgllen = 1;

        /*
         * allocate the reply post queue
         */
        if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
            &mpt->m_dma_post_queue_hdl, &mpt->m_acc_post_queue_hdl, &memp,
            mem_size, &cookie) == FALSE) {
                return (DDI_FAILURE);
        }

        /*
         * Store the reply descriptor post queue memory address.  This chip
         * uses this address to write to the reply descriptor post queue.  The
         * second address is the address mpt uses to manage the queue.
         */
        mpt->m_post_queue_dma_addr = cookie.dmac_laddress;
        mpt->m_post_queue = memp;

        /*
```

```
	 * Clear the reply post queue memory.
	 */
	bzero(mpt->m_post_queue, mem_size);

	return (DDI_SUCCESS);
}

static void
mptsas_alloc_reply_args(mptsas_t *mpt)
{
	if (mpt->m_replyh_args == NULL) {
		mpt->m_replyh_args = kmem_zalloc(sizeof (m_replyh_arg_t) *
		    mpt->m_max_replies, KM_SLEEP);
	}
}

static int
mptsas_alloc_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	mptsas_cache_frames_t	*frames = NULL;
	if (cmd->cmd_extra_frames == NULL) {
		frames = kmem_cache_alloc(mpt->m_cache_frames, KM_NOSLEEP);
		if (frames == NULL) {
			return (DDI_FAILURE);
		}
		cmd->cmd_extra_frames = frames;
	}
	return (DDI_SUCCESS);
}

static void
mptsas_free_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	if (cmd->cmd_extra_frames) {
		kmem_cache_free(mpt->m_cache_frames,
		    (void *)cmd->cmd_extra_frames);
		cmd->cmd_extra_frames = NULL;
	}
}

static void
mptsas_cfg_fini(mptsas_t *mpt)
{
	NDBG0(("mptsas_cfg_fini"));
	ddi_regs_map_free(&mpt->m_datap);
}

static void
mptsas_hba_fini(mptsas_t *mpt)
{
	NDBG0(("mptsas_hba_fini"));

	/*
	 * Free up any allocated memory
	 */
	mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
	    &mpt->m_acc_req_frame_hdl);

	mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
	    &mpt->m_acc_reply_frame_hdl);

	mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
	    &mpt->m_acc_free_queue_hdl);

	mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
	    &mpt->m_acc_post_queue_hdl);
```

```
	if (mpt->m_replyh_args != NULL) {
		kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
		    * mpt->m_max_replies);
	}
}

static int
mptsas_name_child(dev_info_t *lun_dip, char *name, int len)
{
	int		lun = 0;
	char		*sas_wwn = NULL;
	int		phynum = -1;
	int		reallen = 0;

	/* Get the target num */
	lun = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip, DDI_PROP_DONTPASS,
	    LUN_PROP, 0);

	if ((phynum = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip,
	    DDI_PROP_DONTPASS, "sata-phy", -1)) != -1) {
		/*
		 * Stick in the address of form "pPHY,LUN"
		 */
		reallen = snprintf(name, len, "p%x,%x", phynum, lun);
	} else if (ddi_prop_lookup_string(DDI_DEV_T_ANY, lun_dip,
	    DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &sas_wwn)
	    == DDI_PROP_SUCCESS) {
		/*
		 * Stick in the address of the form "wWWN,LUN"
		 */
		reallen = snprintf(name, len, "%s,%x", sas_wwn, lun);
		ddi_prop_free(sas_wwn);
	} else {
		return (DDI_FAILURE);
	}

	ASSERT(reallen < len);
	if (reallen >= len) {
		mptsas_log(0, CE_WARN, "!mptsas_get_name: name parameter "
		    "length too small, it needs to be %d bytes", reallen + 1);
	}
	return (DDI_SUCCESS);
}

/*
 * tran_tgt_init(9E) - target device instance initialization
 */
static int
mptsas_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
{
#ifndef __lock_lint
	_NOTE(ARGUNUSED(hba_tran))
#endif

	/*
	 * At this point, the scsi_device structure already exists
	 * and has been initialized.
	 *
	 * Use this function to allocate target-private data structures,
	 * if needed by this HBA.  Add revised flow-control and queue
	 * properties for child here, if desired and if you can tell they
	 * support tagged queueing by now.
	 */
	mptsas_t		*mpt;
```

```
        int                     lun = sd->sd_address.a_lun;
        mdi_pathinfo_t          *pip = NULL;
        mptsas_tgt_private_t    *tgt_private = NULL;
        mptsas_target_t         *ptgt = NULL;
        char                    *psas_wwn = NULL;
        mptsas_phymask_t        phymask = 0;
        uint64_t                sas_wwn = 0;
        mptsas_target_addr_t    addr;
        mpt = SDEV2MPT(sd);

        ASSERT(scsi_hba_iport_unit_address(hba_dip) != 0);

        NDBG0(("mptsas_scsi_tgt_init: hbadip=0x%p tgtdip=0x%p lun=%d",
            (void *)hba_dip, (void *)tgt_dip, lun));

        if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
                (void) ndi_merge_node(tgt_dip, mptsas_name_child);
                ddi_set_name_addr(tgt_dip, NULL);
                return (DDI_FAILURE);
        }
        /*
         * phymask is 0 means the virtual port for RAID
         */
        phymask = (mptsas_phymask_t)ddi_prop_get_int(DDI_DEV_T_ANY, hba_dip, 0,
            "phymask", 0);
        if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
                if ((pip = (void *)(sd->sd_private)) == NULL) {
                        /*
                         * Very bad news if this occurs. Somehow scsi_vhci has
                         * lost the pathinfo node for this target.
                         */
                        return (DDI_NOT_WELL_FORMED);
                }

                if (mdi_prop_lookup_int(pip, LUN_PROP, &lun) !=
                    DDI_PROP_SUCCESS) {
                        mptsas_log(mpt, CE_WARN, "Get lun property failed\n");
                        return (DDI_FAILURE);
                }

                if (mdi_prop_lookup_string(pip, SCSI_ADDR_PROP_TARGET_PORT,
                    &psas_wwn) == MDI_SUCCESS) {
                        if (scsi_wwnstr_to_wwn(psas_wwn, &sas_wwn)) {
                                sas_wwn = 0;
                        }
                        (void) mdi_prop_free(psas_wwn);
                }
        } else {
                lun = ddi_prop_get_int(DDI_DEV_T_ANY, tgt_dip,
                    DDI_PROP_DONTPASS, LUN_PROP, 0);
                if (ddi_prop_lookup_string(DDI_DEV_T_ANY, tgt_dip,
                    DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &psas_wwn) ==
                    DDI_PROP_SUCCESS) {
                        if (scsi_wwnstr_to_wwn(psas_wwn, &sas_wwn)) {
                                sas_wwn = 0;
                        }
                        ddi_prop_free(psas_wwn);
                } else {
                        sas_wwn = 0;
                }
        }

        ASSERT((sas_wwn != 0) || (phymask != 0));
        addr.mta_wwn = sas_wwn;
        addr.mta_phymask = phymask;
        mutex_enter(&mpt->m_mutex);
```

```
        ptgt = refhash_lookup(mpt->m_targets, &addr);
        mutex_exit(&mpt->m_mutex);
        if (ptgt == NULL) {
                mptsas_log(mpt, CE_WARN, "!tgt_init: target doesn’t exist or "
                    "gone already! phymask:%x, saswwn %"PRIx64, phymask,
                    sas_wwn);
                return (DDI_FAILURE);
        }
        if (hba_tran->tran_tgt_private == NULL) {
                tgt_private = kmem_zalloc(sizeof (mptsas_tgt_private_t),
                    KM_SLEEP);
                tgt_private->t_lun = lun;
                tgt_private->t_private = ptgt;
                hba_tran->tran_tgt_private = tgt_private;
        }

        if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
                return (DDI_SUCCESS);
        }
        mutex_enter(&mpt->m_mutex);

        if (ptgt->m_deviceinfo &
            (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
            MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
                uchar_t *inq89 = NULL;
                int inq89_len = 0x238;
                int reallen = 0;
                int rval = 0;
                struct sata_id *sid = NULL;
                char model[SATA_ID_MODEL_LEN + 1];
                char fw[SATA_ID_FW_LEN + 1];
                char *vid, *pid;
                int i;

                mutex_exit(&mpt->m_mutex);
                /*
                 * According SCSI/ATA Translation -2 (SAT-2) revision 01a
                 * chapter 12.4.2 VPD page 89h includes 512 bytes ATA IDENTIFY
                 * DEVICE data or ATA IDENTIFY PACKET DEVICE data.
                 */
                inq89 = kmem_zalloc(inq89_len, KM_SLEEP);
                rval = mptsas_inquiry(mpt, ptgt, 0, 0x89,
                    inq89, inq89_len, &reallen, 1);

                if (rval != 0) {
                        if (inq89 != NULL) {
                                kmem_free(inq89, inq89_len);
                        }

                        mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
                            "0x89 for SATA target:%x failed!", ptgt->m_devhdl);
                        return (DDI_SUCCESS);
                }
                sid = (void *)(&inq89[60]);

                swab(sid->ai_model, model, SATA_ID_MODEL_LEN);
                swab(sid->ai_fw, fw, SATA_ID_FW_LEN);

                model[SATA_ID_MODEL_LEN] = 0;
                fw[SATA_ID_FW_LEN] = 0;

                /*
                 * split model into into vid/pid
                 */
                for (i = 0, pid = model; i < SATA_ID_MODEL_LEN; i++, pid++)
                        if ((*pid == ’ ’) || (*pid == ’\t’))
```

```
                                break;
                if (i < SATA_ID_MODEL_LEN) {
                        vid = model;
                        /*
                         * terminate vid, establish pid
                         */
                        *pid++ = 0;
                } else {
                        /*
                         * vid will stay "ATA     ", the rule is same
                         * as sata framework implementation.
                         */
                        vid = NULL;
                        /*
                         * model is all pid
                         */
                        pid = model;
                }

                /*
                 * override SCSA "inquiry-*" properties
                 */
                if (vid)
                        (void) scsi_device_prop_update_inqstring(sd,
                            INQUIRY_VENDOR_ID, vid, strlen(vid));
                if (pid)
                        (void) scsi_device_prop_update_inqstring(sd,
                            INQUIRY_PRODUCT_ID, pid, strlen(pid));
                (void) scsi_device_prop_update_inqstring(sd,
                    INQUIRY_REVISION_ID, fw, strlen(fw));

                if (inq89 != NULL) {
                        kmem_free(inq89, inq89_len);
                }
        } else {
                mutex_exit(&mpt->m_mutex);
        }

        return (DDI_SUCCESS);
}
/*
 * tran_tgt_free(9E) - target device instance deallocation
 */
static void
mptsas_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(hba_dip, tgt_dip, hba_tran, sd))
#endif

        mptsas_tgt_private_t    *tgt_private = hba_tran->tran_tgt_private;

        if (tgt_private != NULL) {
                kmem_free(tgt_private, sizeof (mptsas_tgt_private_t));
                hba_tran->tran_tgt_private = NULL;
        }
}

/*
 * scsi_pkt handling
 *
 * Visible to the external world via the transport structure.
 */

/*
```

```
 * Notes:
 *      - transport the command to the addressed SCSI target/lun device
 *      - normal operation is to schedule the command to be transported,
 *        and return TRAN_ACCEPT if this is successful.
 *      - if NO_INTR, tran_start must poll device for command completion
 */
static int
mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(ap))
#endif
        mptsas_t        *mpt = PKT2MPT(pkt);
        mptsas_cmd_t    *cmd = PKT2CMD(pkt);
        int             rval;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

        NDBG1(("mptsas_scsi_start: pkt=0x%p", (void *)pkt));
        ASSERT(ptgt);
        if (ptgt == NULL)
                return (TRAN_FATAL_ERROR);

        /*
         * prepare the pkt before taking mutex.
         */
        rval = mptsas_prepare_pkt(cmd);
        if (rval != TRAN_ACCEPT) {
                return (rval);
        }

        /*
         * Send the command to target/lun, however your HBA requires it.
         * If busy, return TRAN_BUSY; if there’s some other formatting error
         * in the packet, return TRAN_BADPKT; otherwise, fall through to the
         * return of TRAN_ACCEPT.
         *
         * Remember that access to shared resources, including the mptsas_t
         * data structure and the HBA hardware registers, must be protected
         * with mutexes, here and everywhere.
         *
         * Also remember that at interrupt time, you’ll get an argument
         * to the interrupt handler which is a pointer to your mptsas_t
         * structure; you’ll have to remember which commands are outstanding
         * and which scsi_pkt is the currently-running command so the
         * interrupt handler can refer to the pkt to set completion
         * status, call the target driver back through pkt_comp, etc.
         *
         * If the instance lock is held by other thread, don’t spin to wait
         * for it. Instead, queue the cmd and next time when the instance lock
         * is not held, accept all the queued cmd. A extra tx_waitq is
         * introduced to protect the queue.
         *
         * The polled cmd will not be queud and accepted as usual.
         *
         * Under the tx_waitq mutex, record whether a thread is draining
         * the tx_waitq.  An IO requesting thread that finds the instance
         * mutex contended appends to the tx_waitq and while holding the
         * tx_wait mutex, if the draining flag is not set, sets it and then
         * proceeds to spin for the instance mutex. This scheme ensures that
         * the last cmd in a burst be processed.
         *
         * we enable this feature only when the helper threads are enabled,
         * at which we think the loads are heavy.
         *
         * per instance mutex m_tx_waitq_mutex is introduced to protect the
         * m_tx_waitqtail, m_tx_waitq, m_tx_draining.
```

```c
         */

        if (mpt->m_doneq_thread_n) {
                if (mutex_tryenter(&mpt->m_mutex) != 0) {
                        rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
                        mutex_exit(&mpt->m_mutex);
                } else if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
                        mutex_enter(&mpt->m_mutex);
                        rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
                        mutex_exit(&mpt->m_mutex);
                } else {
                        mutex_enter(&mpt->m_tx_waitq_mutex);
                        /*
                         * ptgt->m_dr_flag is protected by m_mutex or
                         * m_tx_waitq_mutex. In this case, m_tx_waitq_mutex
                         * is acquired.
                         */
                        if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
                                if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
                                        /*
                                         * The command should be allowed to
                                         * retry by returning TRAN_BUSY to
                                         * to stall the I/O's which come from
                                         * scsi_vhci since the device/path is
                                         * in unstable state now.
                                         */
                                        mutex_exit(&mpt->m_tx_waitq_mutex);
                                        return (TRAN_BUSY);
                                } else {
                                        /*
                                         * The device is offline, just fail the
                                         * command by returning
                                         * TRAN_FATAL_ERROR.
                                         */
                                        mutex_exit(&mpt->m_tx_waitq_mutex);
                                        return (TRAN_FATAL_ERROR);
                                }
                        }
                        if (mpt->m_tx_draining) {
                                cmd->cmd_flags |= CFLAG_TXQ;
                                *mpt->m_tx_waitqtail = cmd;
                                mpt->m_tx_waitqtail = &cmd->cmd_linkp;
                                mutex_exit(&mpt->m_tx_waitq_mutex);
                        } else { /* drain the queue */
                                mpt->m_tx_draining = 1;
                                mutex_exit(&mpt->m_tx_waitq_mutex);
                                mutex_enter(&mpt->m_mutex);
                                rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
                                mutex_exit(&mpt->m_mutex);
                        }
                }
        } else {
                mutex_enter(&mpt->m_mutex);
                /*
                 * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
                 * in this case, m_mutex is acquired.
                 */
                if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
                        if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
                                /*
                                 * commands should be allowed to retry by
                                 * returning TRAN_BUSY to stall the I/O's
                                 * which come from scsi_vhci since the device/
                                 * path is in unstable state now.
                                 */
                                mutex_exit(&mpt->m_mutex);
```

```c
                                return (TRAN_BUSY);
                        } else {
                                /*
                                 * The device is offline, just fail the
                                 * command by returning TRAN_FATAL_ERROR.
                                 */
                                mutex_exit(&mpt->m_mutex);
                                return (TRAN_FATAL_ERROR);
                        }
                }
                rval = mptsas_accept_pkt(mpt, cmd);
                mutex_exit(&mpt->m_mutex);
        }

        return (rval);
}

/*
 * Accept all the queued cmds(if any) before accept the current one.
 */
static int
mptsas_accept_txwq_and_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        int rval;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

        ASSERT(mutex_owned(&mpt->m_mutex));
        /*
         * The call to mptsas_accept_tx_waitq() must always be performed
         * because that is where mpt->m_tx_draining is cleared.
         */
        mutex_enter(&mpt->m_tx_waitq_mutex);
        mptsas_accept_tx_waitq(mpt);
        mutex_exit(&mpt->m_tx_waitq_mutex);
        /*
         * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
         * in this case, m_mutex is acquired.
         */
        if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
                if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
                        /*
                         * The command should be allowed to retry by returning
                         * TRAN_BUSY to stall the I/O's which come from
                         * scsi_vhci since the device/path is in unstable state
                         * now.
                         */
                        return (TRAN_BUSY);
                } else {
                        /*
                         * The device is offline, just fail the command by
                         * return TRAN_FATAL_ERROR.
                         */
                        return (TRAN_FATAL_ERROR);
                }
        }
        rval = mptsas_accept_pkt(mpt, cmd);

        return (rval);
}

static int
mptsas_accept_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        int             rval = TRAN_ACCEPT;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
        NDBG1(("mptsas_accept_pkt: cmd=0x%p", (void *)cmd));

        ASSERT(mutex_owned(&mpt->m_mutex));

        if ((cmd->cmd_flags & CFLAG_PREPARED) == 0) {
                rval = mptsas_prepare_pkt(cmd);
                if (rval != TRAN_ACCEPT) {
                        cmd->cmd_flags &= ~CFLAG_TRANFLAG;
                        return (rval);
                }
        }

        /*
         * reset the throttle if we were draining
         */
        if ((ptgt->m_t_ncmds == 0) &&
            (ptgt->m_t_throttle == DRAIN_THROTTLE)) {
                NDBG23(("reset throttle"));
                ASSERT(ptgt->m_reset_delay == 0);
                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
        }

        /*
         * If HBA is being reset, the DevHandles are being re-initialized,
         * which means that they could be invalid even if the target is still
         * attached.  Check if being reset and if DevHandle is being
         * re-initialized.  If this is the case, return BUSY so the I/O can be
         * retried later.
         */
        if ((ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) && mpt->m_in_reset) {
                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
                if (cmd->cmd_flags & CFLAG_TXQ) {
                        mptsas_doneq_add(mpt, cmd);
                        mptsas_doneq_empty(mpt);
                        return (rval);
                } else {
                        return (TRAN_BUSY);
                }
        }

        /*
         * If device handle has already been invalidated, just
         * fail the command. In theory, command from scsi_vhci
         * client is impossible send down command with invalid
         * devhdl since devhdl is set after path offline, target
         * driver is not suppose to select a offlined path.
         */
        if (ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) {
                NDBG3(("rejecting command, it might because invalid devhdl "
                    "request."));
                mptsas_set_pkt_reason(mpt, cmd, CMD_DEV_GONE, STAT_TERMINATED);
                if (cmd->cmd_flags & CFLAG_TXQ) {
                        mptsas_doneq_add(mpt, cmd);
                        mptsas_doneq_empty(mpt);
                        return (rval);
                } else {
                        return (TRAN_FATAL_ERROR);
                }
        }
        /*
         * The first case is the normal case.  mpt gets a command from the
         * target driver and starts it.
         * Since SMID 0 is reserved and the TM slot is reserved, the actual max
         * commands is m_max_requests - 2.
         */
        if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
```

```
            (ptgt->m_t_throttle > HOLD_THROTTLE) &&
            (ptgt->m_t_ncmds < ptgt->m_t_throttle) &&
            (ptgt->m_reset_delay == 0) &&
            (ptgt->m_t_nwait == 0) &&
            ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0)) {
                if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                        (void) mptsas_start_cmd(mpt, cmd);
                } else {
                        mptsas_waitq_add(mpt, cmd);
                }
        } else {
                /*
                 * Add this pkt to the work queue
                 */
                mptsas_waitq_add(mpt, cmd);

                if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
                        (void) mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);

                        /*
                         * Only flush the doneq if this is not a TM
                         * cmd.  For TM cmds the flushing of the
                         * doneq will be done in those routines.
                         */
                        if ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
                                mptsas_doneq_empty(mpt);
                        }
                }
        }
        return (rval);
}

int
mptsas_save_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        mptsas_slots_t *slots = mpt->m_active;
        uint_t slot, start_rotor;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

        ASSERT(MUTEX_HELD(&mpt->m_mutex));

        /*
         * Account for reserved TM request slot and reserved SMID of 0.
         */
        ASSERT(slots->m_n_normal == (mpt->m_max_requests - 2));

        /*
         * Find the next available slot, beginning at m_rotor.  If no slot is
         * available, we'll return FALSE to indicate that.  This mechanism
         * considers only the normal slots, not the reserved slot 0 nor the
         * task management slot m_n_normal + 1.  The rotor is left to point to
         * the normal slot after the one we select, unless we select the last
         * normal slot in which case it returns to slot 1.
         */
        start_rotor = slots->m_rotor;
        do {
                slot = slots->m_rotor++;
                if (slots->m_rotor > slots->m_n_normal)
                        slots->m_rotor = 1;

                if (slots->m_rotor == start_rotor)
                        break;
        } while (slots->m_slot[slot] != NULL);

        if (slots->m_slot[slot] != NULL)
                return (FALSE);
```

```
	ASSERT(slot != 0 && slot <= slots->m_n_normal);

	cmd->cmd_slot = slot;
	slots->m_slot[slot] = cmd;
	mpt->m_ncmds++;

	/*
	 * only increment per target ncmds if this is not a
	 * command that has no target associated with it (i.e. a
	 * event acknoledgment)
	 */
	if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
		/*
		 * Expiration time is set in mptsas_start_cmd
		 */
		ptgt->m_t_ncmds++;
		cmd->cmd_active_expiration = 0;
	} else {
		/*
		 * Initialize expiration time for passthrough commands,
		 */
		cmd->cmd_active_expiration = gethrtime() +
		    (hrtime_t)cmd->cmd_pkt->pkt_time * NANOSEC;
	}
	return (TRUE);
}

/*
 * prepare the pkt:
 * the pkt may have been resubmitted or just reused so
 * initialize some fields and do some checks.
 */
static int
mptsas_prepare_pkt(mptsas_cmd_t *cmd)
{
	struct scsi_pkt *pkt = CMD2PKT(cmd);

	NDBG1(("mptsas_prepare_pkt: cmd=0x%p", (void *)cmd));

	/*
	 * Reinitialize some fields that need it; the packet may
	 * have been resubmitted
	 */
	pkt->pkt_reason = CMD_CMPLT;
	pkt->pkt_state = 0;
	pkt->pkt_statistics = 0;
	pkt->pkt_resid = 0;
	cmd->cmd_age = 0;
	cmd->cmd_pkt_flags = pkt->pkt_flags;

	/*
	 * zero status byte.
	 */
	*(pkt->pkt_scbp) = 0;

	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		pkt->pkt_resid = cmd->cmd_dmacount;

		/*
		 * consistent packets need to be sync'ed first
		 * (only for data going out)
		 */
		if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
		    (cmd->cmd_flags & CFLAG_DMASEND)) {
			(void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
```

```
			    DDI_DMA_SYNC_FORDEV);
		}
	}

	cmd->cmd_flags =
	    (cmd->cmd_flags & ~(CFLAG_TRANFLAG)) |
	    CFLAG_PREPARED | CFLAG_IN_TRANSPORT;

	return (TRAN_ACCEPT);
}

/*
 * tran_init_pkt(9E) - allocate scsi_pkt(9S) for command
 *
 * One of three possibilities:
 *	- allocate scsi_pkt
 *	- allocate scsi_pkt and DMA resources
 *	- allocate DMA resources to an already-allocated pkt
 */
static struct scsi_pkt *
mptsas_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
    struct buf *bp, int cmdlen, int statuslen, int tgtlen, int flags,
    int (*callback)(), caddr_t arg)
{
	mptsas_cmd_t		*cmd, *new_cmd;
	mptsas_t		*mpt = ADDR2MPT(ap);
	int			failure = 1;
	uint_t			oldcookiec;
	mptsas_target_t		*ptgt = NULL;
	int			rval;
	mptsas_tgt_private_t	*tgt_private;
	int			kf;

	kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;

	tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
	    tran_tgt_private;
	ASSERT(tgt_private != NULL);
	if (tgt_private == NULL) {
		return (NULL);
	}
	ptgt = tgt_private->t_private;
	ASSERT(ptgt != NULL);
	if (ptgt == NULL)
		return (NULL);
	ap->a_target = ptgt->m_devhdl;
	ap->a_lun = tgt_private->t_lun;

	ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);
#ifdef MPTSAS_TEST_EXTRN_ALLOC
	statuslen *= 100; tgtlen *= 4;
#endif
	NDBG3(("mptsas_scsi_init_pkt:\n"
	    "\ttgt=%d in=0x%p bp=0x%p clen=%d slen=%d tlen=%d flags=%x",
	    ap->a_target, (void *)pkt, (void *)bp,
	    cmdlen, statuslen, tgtlen, flags));

	/*
	 * Allocate the new packet.
	 */
	if (pkt == NULL) {
		ddi_dma_handle_t	save_dma_handle;
		ddi_dma_handle_t	save_arq_dma_handle;
		struct buf		*save_arq_bp;
		ddi_dma_cookie_t	save_arqcookie;
		cmd = kmem_cache_alloc(mpt->m_kmem_cache, kf);

		if (cmd) {
			save_dma_handle = cmd->cmd_dmahandle;
			save_arq_dma_handle = cmd->cmd_arqhandle;
			save_arq_bp = cmd->cmd_arq_buf;
			save_arqcookie = cmd->cmd_arqcookie;
			bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
			cmd->cmd_dmahandle = save_dma_handle;
			cmd->cmd_arqhandle = save_arq_dma_handle;
			cmd->cmd_arq_buf = save_arq_bp;
			cmd->cmd_arqcookie = save_arqcookie;

			pkt = (void *)((uchar_t *)cmd +
			    sizeof (struct mptsas_cmd));
			pkt->pkt_ha_private = (opaque_t)cmd;
			pkt->pkt_address = *ap;
			pkt->pkt_private = (opaque_t)cmd->cmd_pkt_private;
			pkt->pkt_scbp = (opaque_t)&cmd->cmd_scb;
			pkt->pkt_cdbp = (opaque_t)&cmd->cmd_cdb;
			cmd->cmd_pkt = (struct scsi_pkt *)pkt;
			cmd->cmd_cdblen = (uchar_t)cmdlen;
			cmd->cmd_scblen = statuslen;
			cmd->cmd_rqslen = SENSE_LENGTH;
			cmd->cmd_tgt_addr = ptgt;
			failure = 0;
		}

		if (failure || (cmdlen > sizeof (cmd->cmd_cdb)) ||
		    (tgtlen > PKT_PRIV_LEN) ||
		    (statuslen > EXTCMDS_STATUS_SIZE)) {
			if (failure == 0) {
				/*
				 * if extern alloc fails, all will be
				 * deallocated, including cmd
				 */
				failure = mptsas_pkt_alloc_extern(mpt, cmd,
				    cmdlen, tgtlen, statuslen, kf);
			}
			if (failure) {
				/*
				 * if extern allocation fails, it will
				 * deallocate the new pkt as well
				 */
				return (NULL);
			}
		}
		new_cmd = cmd;

	} else {
		cmd = PKT2CMD(pkt);
		new_cmd = NULL;
	}


	/* grab cmd->cmd_cookiec here as oldcookiec */

	oldcookiec = cmd->cmd_cookiec;

	/*
	 * If the dma was broken up into PARTIAL transfers cmd_nwin will be
	 * greater than 0 and we'll need to grab the next dma window
	 */
	/*
	 * SLM-not doing extra command frame right now; may add later
	 */
```

```

	if (cmd->cmd_nwin > 0) {

		/*
		 * Make sure we havn't gone past the the total number
		 * of windows
		 */
		if (++cmd->cmd_winindex >= cmd->cmd_nwin) {
			return (NULL);
		}
		if (ddi_dma_getwin(cmd->cmd_dmahandle, cmd->cmd_winindex,
		    &cmd->cmd_dma_offset, &cmd->cmd_dma_len,
		    &cmd->cmd_cookie, &cmd->cmd_cookiec) == DDI_FAILURE) {
			return (NULL);
		}
		goto get_dma_cookies;
	}


	if (flags & PKT_XARQ) {
		cmd->cmd_flags |= CFLAG_XARQ;
	}

	/*
	 * DMA resource allocation.  This version assumes your
	 * HBA has some sort of bus-mastering or onboard DMA capability, with a
	 * scatter-gather list of length MPTSAS_MAX_DMA_SEGS, as given in the
	 * ddi_dma_attr_t structure and passed to scsi_impl_dmaget.
	 */
	if (bp && (bp->b_bcount != 0) &&
	    (cmd->cmd_flags & CFLAG_DMAVALID) == 0) {

		int	cnt, dma_flags;
		mptti_t *dmap;		/* ptr to the S/G list */

		/*
		 * Set up DMA memory and position to the next DMA segment.
		 */
		ASSERT(cmd->cmd_dmahandle != NULL);

		if (bp->b_flags & B_READ) {
			dma_flags = DDI_DMA_READ;
			cmd->cmd_flags &= ~CFLAG_DMASEND;
		} else {
			dma_flags = DDI_DMA_WRITE;
			cmd->cmd_flags |= CFLAG_DMASEND;
		}
		if (flags & PKT_CONSISTENT) {
			cmd->cmd_flags |= CFLAG_CMDIOPB;
			dma_flags |= DDI_DMA_CONSISTENT;
		}

		if (flags & PKT_DMA_PARTIAL) {
			dma_flags |= DDI_DMA_PARTIAL;
		}

		/*
		 * workaround for byte hole issue on psycho and
		 * schizo pre 2.1
		 */
		if ((bp->b_flags & B_READ) && ((bp->b_flags &
		    (B_PAGEIO|B_REMAPPED)) != B_PAGEIO) &&
		    ((uintptr_t)bp->b_un.b_addr & 0x7)) {
			dma_flags |= DDI_DMA_CONSISTENT;
		}
```

```
                rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
                    dma_flags, callback, arg,
                    &cmd->cmd_cookie, &cmd->cmd_cookiec);
                if (rval == DDI_DMA_PARTIAL_MAP) {
                        (void) ddi_dma_numwin(cmd->cmd_dmahandle,
                            &cmd->cmd_nwin);
                        cmd->cmd_winindex = 0;
                        (void) ddi_dma_getwin(cmd->cmd_dmahandle,
                            cmd->cmd_winindex, &cmd->cmd_dma_offset,
                            &cmd->cmd_dma_len, &cmd->cmd_cookie,
                            &cmd->cmd_cookiec);
                } else if (rval && (rval != DDI_DMA_MAPPED)) {
                        switch (rval) {
                        case DDI_DMA_NORESOURCES:
                                bioerror(bp, 0);
                                break;
                        case DDI_DMA_BADATTR:
                        case DDI_DMA_NOMAPPING:
                                bioerror(bp, EFAULT);
                                break;
                        case DDI_DMA_TOOBIG:
                        default:
                                bioerror(bp, EINVAL);
                                break;
                        }
                        cmd->cmd_flags &= ~CFLAG_DMAVALID;
                        if (new_cmd) {
                                mptsas_scsi_destroy_pkt(ap, pkt);
                        }
                        return ((struct scsi_pkt *)NULL);
                }

get_dma_cookies:
                cmd->cmd_flags |= CFLAG_DMAVALID;
                ASSERT(cmd->cmd_cookiec > 0);

                if (cmd->cmd_cookiec > MPTSAS_MAX_CMD_SEGS) {
                        mptsas_log(mpt, CE_NOTE, "large cookiec received %d\n",
                            cmd->cmd_cookiec);
                        bioerror(bp, EINVAL);
                        if (new_cmd) {
                                mptsas_scsi_destroy_pkt(ap, pkt);
                        }
                        return ((struct scsi_pkt *)NULL);
                }

                /*
                 * Allocate extra SGL buffer if needed.
                 */
                if ((cmd->cmd_cookiec > MPTSAS_MAX_FRAME_SGES64(mpt)) &&
                    (cmd->cmd_extra_frames == NULL)) {
                        if (mptsas_alloc_extra_sgl_frame(mpt, cmd) ==
                            DDI_FAILURE) {
                                mptsas_log(mpt, CE_WARN, "MPT SGL mem alloc "
                                    "failed");
                                bioerror(bp, ENOMEM);
                                if (new_cmd) {
                                        mptsas_scsi_destroy_pkt(ap, pkt);
                                }
                                return ((struct scsi_pkt *)NULL);
                        }
                }

                /*
                 * Always use scatter-gather transfer
                 * Use the loop below to store physical addresses of
```

```
                 * DMA segments, from the DMA cookies, into your HBA’s
                 * scatter-gather list.
                 * We need to ensure we have enough kmem alloc’d
                 * for the sg entries since we are no longer using an
                 * array inside mptsas_cmd_t.
                 *
                 * We check cmd->cmd_cookiec against oldcookiec so
                 * the scatter-gather list is correctly allocated
                 */

                if (oldcookiec != cmd->cmd_cookiec) {
                        if (cmd->cmd_sg != (mptti_t *)NULL) {
                                kmem_free(cmd->cmd_sg, sizeof (mptti_t) *
                                    oldcookiec);
                                cmd->cmd_sg = NULL;
                        }
                }

                if (cmd->cmd_sg == (mptti_t *)NULL) {
                        cmd->cmd_sg = kmem_alloc((size_t)(sizeof (mptti_t)*
                            cmd->cmd_cookiec), kf);

                        if (cmd->cmd_sg == (mptti_t *)NULL) {
                                mptsas_log(mpt, CE_WARN,
                                    "unable to kmem_alloc enough memory "
                                    "for scatter/gather list");
                /*
                 * if we have an ENOMEM condition we need to behave
                 * the same way as the rest of this routine
                 */

                                bioerror(bp, ENOMEM);
                                if (new_cmd) {
                                        mptsas_scsi_destroy_pkt(ap, pkt);
                                }
                                return ((struct scsi_pkt *)NULL);
                        }
                }

                dmap = cmd->cmd_sg;

                ASSERT(cmd->cmd_cookie.dmac_size != 0);

                /*
                 * store the first segment into the S/G list
                 */
                dmap->count = cmd->cmd_cookie.dmac_size;
                dmap->addr.address64.Low = (uint32_t)
                    (cmd->cmd_cookie.dmac_laddress & 0xffffffffull);
                dmap->addr.address64.High = (uint32_t)
                    (cmd->cmd_cookie.dmac_laddress >> 32);

                /*
                 * dmacount counts the size of the dma for this window
                 * (if partial dma is being used).  totaldmacount
                 * keeps track of the total amount of dma we have
                 * transferred for all the windows (needed to calculate
                 * the resid value below).
                 */
                cmd->cmd_dmacount = cmd->cmd_cookie.dmac_size;
                cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;

                /*
                 * We already stored the first DMA scatter gather segment,
                 * start at 1 if we need to store more.
                 */
```

```
		for (cnt = 1; cnt < cmd->cmd_cookiec; cnt++) {
			/*
			 * Get next DMA cookie
			 */
			ddi_dma_nextcookie(cmd->cmd_dmahandle,
			    &cmd->cmd_cookie);
			dmap++;

			cmd->cmd_dmacount += cmd->cmd_cookie.dmac_size;
			cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;

			/*
			 * store the segment parms into the S/G list
			 */
			dmap->count = cmd->cmd_cookie.dmac_size;
			dmap->addr.address64.Low = (uint32_t)
			    (cmd->cmd_cookie.dmac_laddress & 0xffffffffull);
			dmap->addr.address64.High = (uint32_t)
			    (cmd->cmd_cookie.dmac_laddress >> 32);
		}

		/*
		 * If this was partially allocated we set the resid
		 * the amount of data NOT transferred in this window
		 * If there is only one window, the resid will be 0
		 */
		pkt->pkt_resid = (bp->b_bcount - cmd->cmd_totaldmacount);
		NDBG3(("mptsas_scsi_init_pkt: cmd_dmacount=%d.",
		    cmd->cmd_dmacount));
	}
	return (pkt);
}

/*
 * tran_destroy_pkt(9E) - scsi_pkt(9s) deallocation
 *
 * Notes:
 *	- also frees DMA resources if allocated
 *	- implicit DMA synchonization
 */
static void
mptsas_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
{
	mptsas_cmd_t	*cmd = PKT2CMD(pkt);
	mptsas_t	*mpt = ADDR2MPT(ap);

	NDBG3(("mptsas_scsi_destroy_pkt: target=%d pkt=0x%p",
	    ap->a_target, (void *)pkt));

	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		(void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
		cmd->cmd_flags &= ~CFLAG_DMAVALID;
	}

	if (cmd->cmd_sg) {
		kmem_free(cmd->cmd_sg, sizeof (mptti_t) * cmd->cmd_cookiec);
		cmd->cmd_sg = NULL;
	}

	mptsas_free_extra_sgl_frame(mpt, cmd);

	if ((cmd->cmd_flags &
	    (CFLAG_FREE | CFLAG_CDBEXTERN | CFLAG_PRIVEXTERN |
	    CFLAG_SCBEXTERN)) == 0) {
		cmd->cmd_flags = CFLAG_FREE;
		kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
```

```
	} else {
		mptsas_pkt_destroy_extern(mpt, cmd);
	}
}

/*
 * kmem cache constructor and destructor:
 * When constructing, we bzero the cmd and allocate the dma handle
 * When destructing, just free the dma handle
 */
static int
mptsas_kmem_cache_constructor(void *buf, void *cdrarg, int kmflags)
{
	mptsas_cmd_t		*cmd = buf;
	mptsas_t		*mpt  = cdrarg;
	struct scsi_address	ap;
	uint_t			cookiec;
	ddi_dma_attr_t		arq_dma_attr;
	int			(*callback)(caddr_t);

	callback = (kmflags == KM_SLEEP)? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;

	NDBG4(("mptsas_kmem_cache_constructor"));

	ap.a_hba_tran = mpt->m_tran;
	ap.a_target = 0;
	ap.a_lun = 0;

	/*
	 * allocate a dma handle
	 */
	if ((ddi_dma_alloc_handle(mpt->m_dip, &mpt->m_io_dma_attr, callback,
	    NULL, &cmd->cmd_dmahandle)) != DDI_SUCCESS) {
		cmd->cmd_dmahandle = NULL;
		return (-1);
	}

	cmd->cmd_arq_buf = scsi_alloc_consistent_buf(&ap, (struct buf *)NULL,
	    SENSE_LENGTH, B_READ, callback, NULL);
	if (cmd->cmd_arq_buf == NULL) {
		ddi_dma_free_handle(&cmd->cmd_dmahandle);
		cmd->cmd_dmahandle = NULL;
		return (-1);
	}

	/*
	 * allocate a arq handle
	 */
	arq_dma_attr = mpt->m_msg_dma_attr;
	arq_dma_attr.dma_attr_sgllen = 1;
	if ((ddi_dma_alloc_handle(mpt->m_dip, &arq_dma_attr, callback,
	    NULL, &cmd->cmd_arqhandle)) != DDI_SUCCESS) {
		ddi_dma_free_handle(&cmd->cmd_dmahandle);
		scsi_free_consistent_buf(cmd->cmd_arq_buf);
		cmd->cmd_dmahandle = NULL;
		cmd->cmd_arqhandle = NULL;
		return (-1);
	}

	if (ddi_dma_buf_bind_handle(cmd->cmd_arqhandle,
	    cmd->cmd_arq_buf, (DDI_DMA_READ | DDI_DMA_CONSISTENT),
	    callback, NULL, &cmd->cmd_arqcookie, &cookiec) != DDI_SUCCESS) {
		ddi_dma_free_handle(&cmd->cmd_dmahandle);
		ddi_dma_free_handle(&cmd->cmd_arqhandle);
		scsi_free_consistent_buf(cmd->cmd_arq_buf);
		cmd->cmd_dmahandle = NULL;
```

```
                cmd->cmd_arqhandle = NULL;
                cmd->cmd_arq_buf = NULL;
                return (-1);
        }

        return (0);
}

static void
mptsas_kmem_cache_destructor(void *buf, void *cdrarg)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(cdrarg))
#endif
        mptsas_cmd_t    *cmd = buf;

        NDBG4(("mptsas_kmem_cache_destructor"));

        if (cmd->cmd_arqhandle) {
                (void) ddi_dma_unbind_handle(cmd->cmd_arqhandle);
                ddi_dma_free_handle(&cmd->cmd_arqhandle);
                cmd->cmd_arqhandle = NULL;
        }
        if (cmd->cmd_arq_buf) {
                scsi_free_consistent_buf(cmd->cmd_arq_buf);
                cmd->cmd_arq_buf = NULL;
        }
        if (cmd->cmd_dmahandle) {
                ddi_dma_free_handle(&cmd->cmd_dmahandle);
                cmd->cmd_dmahandle = NULL;
        }
}

static int
mptsas_cache_frames_constructor(void *buf, void *cdrarg, int kmflags)
{
        mptsas_cache_frames_t   *p = buf;
        mptsas_t                *mpt = cdrarg;
        ddi_dma_attr_t          frame_dma_attr;
        size_t                  mem_size, alloc_len;
        ddi_dma_cookie_t        cookie;
        uint_t                  ncookie;
        int (*callback)(caddr_t) = (kmflags == KM_SLEEP)
            ? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;

        frame_dma_attr = mpt->m_msg_dma_attr;
        frame_dma_attr.dma_attr_align = 0x10;
        frame_dma_attr.dma_attr_sgllen = 1;

        if (ddi_dma_alloc_handle(mpt->m_dip, &frame_dma_attr, callback, NULL,
            &p->m_dma_hdl) != DDI_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "Unable to allocate dma handle for"
                    " extra SGL.");
                return (DDI_FAILURE);
        }

        mem_size = (mpt->m_max_request_frames - 1) * mpt->m_req_frame_size;

        if (ddi_dma_mem_alloc(p->m_dma_hdl, mem_size, &mpt->m_dev_acc_attr,
            DDI_DMA_CONSISTENT, callback, NULL, (caddr_t *)&p->m_frames_addr,
            &alloc_len, &p->m_acc_hdl) != DDI_SUCCESS) {
                ddi_dma_free_handle(&p->m_dma_hdl);
                p->m_dma_hdl = NULL;
                mptsas_log(mpt, CE_WARN, "Unable to allocate dma memory for"
                    " extra SGL.");
                return (DDI_FAILURE);
```

```
        }

        if (ddi_dma_addr_bind_handle(p->m_dma_hdl, NULL, p->m_frames_addr,
            alloc_len, DDI_DMA_RDWR | DDI_DMA_CONSISTENT, callback, NULL,
            &cookie, &ncookie) != DDI_DMA_MAPPED) {
                (void) ddi_dma_mem_free(&p->m_acc_hdl);
                ddi_dma_free_handle(&p->m_dma_hdl);
                p->m_dma_hdl = NULL;
                mptsas_log(mpt, CE_WARN, "Unable to bind DMA resources for"
                    " extra SGL");
                return (DDI_FAILURE);
        }

        /*
         * Store the SGL memory address.  This chip uses this
         * address to dma to and from the driver.  The second
         * address is the address mpt uses to fill in the SGL.
         */
        p->m_phys_addr = cookie.dmac_address;

        return (DDI_SUCCESS);
}

static void
mptsas_cache_frames_destructor(void *buf, void *cdrarg)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(cdrarg))
#endif
        mptsas_cache_frames_t   *p = buf;
        if (p->m_dma_hdl != NULL) {
                (void) ddi_dma_unbind_handle(p->m_dma_hdl);
                (void) ddi_dma_mem_free(&p->m_acc_hdl);
                ddi_dma_free_handle(&p->m_dma_hdl);
                p->m_phys_addr = NULL;
                p->m_frames_addr = NULL;
                p->m_dma_hdl = NULL;
                p->m_acc_hdl = NULL;
        }

}

/*
 * allocate and deallocate external pkt space (ie. not part of mptsas_cmd)
 * for non-standard length cdb, pkt_private, status areas
 * if allocation fails, then deallocate all external space and the pkt
 */
/* ARGSUSED */
static int
mptsas_pkt_alloc_extern(mptsas_t *mpt, mptsas_cmd_t *cmd,
    int cmdlen, int tgtlen, int statuslen, int kf)
{
        caddr_t                 cdbp, scbp, tgt;
        int                     (*callback)(caddr_t) = (kf == KM_SLEEP) ?
            DDI_DMA_SLEEP : DDI_DMA_DONTWAIT;
        struct scsi_address     ap;
        size_t                  senselength;
        ddi_dma_attr_t          ext_arq_dma_attr;
        uint_t                  cookiec;

        NDBG3(("mptsas_pkt_alloc_extern: "
            "cmd=0x%p cmdlen=%d tgtlen=%d statuslen=%d kf=%x",
            (void *)cmd, cmdlen, tgtlen, statuslen, kf));

        tgt = cdbp = scbp = NULL;
        cmd->cmd_scblen         = statuslen;
        cmd->cmd_privlen        = (uchar_t)tgtlen;

        if (cmdlen > sizeof (cmd->cmd_cdb)) {
                if ((cdbp = kmem_zalloc((size_t)cmdlen, kf)) == NULL) {
                        goto fail;
                }
                cmd->cmd_pkt->pkt_cdbp = (opaque_t)cdbp;
                cmd->cmd_flags |= CFLAG_CDBEXTERN;
        }
        if (tgtlen > PKT_PRIV_LEN) {
                if ((tgt = kmem_zalloc((size_t)tgtlen, kf)) == NULL) {
                        goto fail;
                }
                cmd->cmd_flags |= CFLAG_PRIVEXTERN;
                cmd->cmd_pkt->pkt_private = tgt;
        }
        if (statuslen > EXTCMDS_STATUS_SIZE) {
                if ((scbp = kmem_zalloc((size_t)statuslen, kf)) == NULL) {
                        goto fail;
                }
                cmd->cmd_flags |= CFLAG_SCBEXTERN;
                cmd->cmd_pkt->pkt_scbp = (opaque_t)scbp;

                /* allocate sense data buf for DMA */

                senselength = statuslen - MPTSAS_GET_ITEM_OFF(
                    struct scsi_arq_status, sts_sensedata);
                cmd->cmd_rqslen = (uchar_t)senselength;

                ap.a_hba_tran = mpt->m_tran;
                ap.a_target = 0;
                ap.a_lun = 0;

                cmd->cmd_ext_arq_buf = scsi_alloc_consistent_buf(&ap,
                    (struct buf *)NULL, senselength, B_READ,
                    callback, NULL);

                if (cmd->cmd_ext_arq_buf == NULL) {
                        goto fail;
                }
                /*
                 * allocate a extern arq handle and bind the buf
                 */
                ext_arq_dma_attr = mpt->m_msg_dma_attr;
                ext_arq_dma_attr.dma_attr_sgllen = 1;
                if ((ddi_dma_alloc_handle(mpt->m_dip,
                    &ext_arq_dma_attr, callback,
                    NULL, &cmd->cmd_ext_arqhandle)) != DDI_SUCCESS) {
                        goto fail;
                }

                if (ddi_dma_buf_bind_handle(cmd->cmd_ext_arqhandle,
                    cmd->cmd_ext_arq_buf, (DDI_DMA_READ | DDI_DMA_CONSISTENT),
                    callback, NULL, &cmd->cmd_ext_arqcookie,
                    &cookiec)
                    != DDI_SUCCESS) {
                        goto fail;
                }
                cmd->cmd_flags |= CFLAG_EXTARQBUFVALID;
        }
        return (0);
fail:
        mptsas_pkt_destroy_extern(mpt, cmd);
        return (1);
}
```

```
/*
 * deallocate external pkt space and deallocate the pkt
 */
static void
mptsas_pkt_destroy_extern(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        NDBG3(("mptsas_pkt_destroy_extern: cmd=0x%p", (void *)cmd));

        if (cmd->cmd_flags & CFLAG_FREE) {
                mptsas_log(mpt, CE_PANIC,
                    "mptsas_pkt_destroy_extern: freeing free packet");
                _NOTE(NOT_REACHED)
                /* NOTREACHED */
        }
        if (cmd->cmd_flags & CFLAG_CDBEXTERN) {
                kmem_free(cmd->cmd_pkt->pkt_cdbp, (size_t)cmd->cmd_cdblen);
        }
        if (cmd->cmd_flags & CFLAG_SCBEXTERN) {
                kmem_free(cmd->cmd_pkt->pkt_scbp, (size_t)cmd->cmd_scblen);
                if (cmd->cmd_flags & CFLAG_EXTARQBUFVALID) {
                        (void) ddi_dma_unbind_handle(cmd->cmd_ext_arqhandle);
                }
                if (cmd->cmd_ext_arqhandle) {
                        ddi_dma_free_handle(&cmd->cmd_ext_arqhandle);
                        cmd->cmd_ext_arqhandle = NULL;
                }
                if (cmd->cmd_ext_arq_buf)
                        scsi_free_consistent_buf(cmd->cmd_ext_arq_buf);
        }
        if (cmd->cmd_flags & CFLAG_PRIVEXTERN) {
                kmem_free(cmd->cmd_pkt->pkt_private, (size_t)cmd->cmd_privlen);
        }
        cmd->cmd_flags = CFLAG_FREE;
        kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
}

/*
 * tran_sync_pkt(9E) - explicit DMA synchronization
 */
/*ARGSUSED*/
static void
mptsas_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
{
        mptsas_cmd_t    *cmd = PKT2CMD(pkt);

        NDBG3(("mptsas_scsi_sync_pkt: target=%d, pkt=0x%p",
            ap->a_target, (void *)pkt));

        if (cmd->cmd_dmahandle) {
                (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
                    (cmd->cmd_flags & CFLAG_DMASEND) ?
                    DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
        }
}

/*
 * tran_dmafree(9E) - deallocate DMA resources allocated for command
 */
/*ARGSUSED*/
static void
mptsas_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
{
        mptsas_cmd_t    *cmd = PKT2CMD(pkt);
        mptsas_t        *mpt = ADDR2MPT(ap);

        NDBG3(("mptsas_scsi_dmafree: target=%d pkt=0x%p",
            ap->a_target, (void *)pkt));

        if (cmd->cmd_flags & CFLAG_DMAVALID) {
                (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
                cmd->cmd_flags &= ~CFLAG_DMAVALID;
        }

        if (cmd->cmd_flags & CFLAG_EXTARQBUFVALID) {
                (void) ddi_dma_unbind_handle(cmd->cmd_ext_arqhandle);
                cmd->cmd_flags &= ~CFLAG_EXTARQBUFVALID;
        }

        mptsas_free_extra_sgl_frame(mpt, cmd);
}

static void
mptsas_pkt_comp(struct scsi_pkt *pkt, mptsas_cmd_t *cmd)
{
        if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
            (!(cmd->cmd_flags & CFLAG_DMASEND))) {
                (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
                    DDI_DMA_SYNC_FORCPU);
        }
        (*pkt->pkt_comp)(pkt);
}

static void
mptsas_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
    ddi_acc_handle_t acc_hdl, uint_t cookiec, uint32_t end_flags)
{
        pMpi2SGESimple64_t      sge;
        mptti_t                 *dmap;
        uint32_t                flags;

        dmap = cmd->cmd_sg;

        sge = (pMpi2SGESimple64_t)(&frame->SGL);
        while (cookiec--) {
                ddi_put32(acc_hdl,
                    &sge->Address.Low, dmap->addr.address64.Low);
                ddi_put32(acc_hdl,
                    &sge->Address.High, dmap->addr.address64.High);
                ddi_put32(acc_hdl, &sge->FlagsLength,
                    dmap->count);
                flags = ddi_get32(acc_hdl, &sge->FlagsLength);
                flags |= ((uint32_t)
                    (MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
                    MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
                    MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
                    MPI2_SGE_FLAGS_SHIFT);

                /*
                 * If this is the last cookie, we set the flags
                 * to indicate so
                 */
                if (cookiec == 0) {
                        flags |= end_flags;
                }
                if (cmd->cmd_flags & CFLAG_DMASEND) {
                        flags |= (MPI2_SGE_FLAGS_HOST_TO_IOC <<
                            MPI2_SGE_FLAGS_SHIFT);
                } else {
                        flags |= (MPI2_SGE_FLAGS_IOC_TO_HOST <<
                            MPI2_SGE_FLAGS_SHIFT);
                }
                ddi_put32(acc_hdl, &sge->FlagsLength, flags);
```

```
                dmap++;
                sge++;
        }
}

static void
mptsas_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
    pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
{
        pMpi2SGESimple64_t      sge;
        pMpi2SGEChain64_t       sgechain;
        uint_t                  cookiec;
        mptti_t                 *dmap;
        uint32_t                flags;

        /*
         * Save the number of entries in the DMA
         * Scatter/Gather list
         */
        cookiec = cmd->cmd_cookiec;

        /*
         * Hereby we start to deal with multiple frames.
         * The process is as follows:
         * 1. Determine how many frames are needed for SGL element
         *    storage; Note that all frames are stored in contiguous
         *    memory space and in 64-bit DMA mode each element is
         *    3 double-words (12 bytes) long.
         * 2. Fill up the main frame. We need to do this separately
         *    since it contains the SCSI IO request header and needs
         *    dedicated processing. Note that the last 4 double-words
         *    of the SCSI IO header is for SGL element storage
         *    (MPI2_SGE_IO_UNION).
         * 3. Fill the chain element in the main frame, so the DMA
         *    engine can use the following frames.
         * 4. Enter a loop to fill the remaining frames. Note that the
         *    last frame contains no chain element.  The remaining
         *    frames go into the mpt SGL buffer allocated on the fly,
         *    not immediately following the main message frame, as in
         *    Gen1.
         * Some restrictions:
         * 1. For 64-bit DMA, the simple element and chain element
         *    are both of 3 double-words (12 bytes) in size, even
         *    though all frames are stored in the first 4G of mem
         *    range and the higher 32-bits of the address are always 0.
         * 2. On some controllers (like the 1064/1068), a frame can
         *    hold SGL elements with the last 1 or 2 double-words
         *    (4 or 8 bytes) un-used. On these controllers, we should
         *    recognize that there's not enough room for another SGL
         *    element and move the sge pointer to the next frame.
         */
        int                     i, j, k, l, frames, sgemax;
        int                     temp;
        uint8_t                 chainflags;
        uint16_t                chainlength;
        mptsas_cache_frames_t   *p;

        /*
         * Sgemax is the number of SGE's that will fit
         * each extra frame and frames is total
         * number of frames we'll need.  1 sge entry per
         * frame is reseverd for the chain element thus the -1 below.
         */
        sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_SGE_SIMPLE64))
            - 1);
        temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;
```

```c
        /*
         * A little check to see if we need to round up the number
         * of frames we need
         */
        if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
            sgemax) > 1) {
                frames = (temp + 1);
        } else {
                frames = temp;
        }
        dmap = cmd->cmd_sg;
        sge = (pMpi2SGESimple64_t)(&frame->SGL);

        /*
         * First fill in the main frame
         */
        j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
        mptsas_sge_mainframe(cmd, frame, acc_hdl, j,
            ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT) <<
            MPI2_SGE_FLAGS_SHIFT));
        dmap += j;
        sge += j;
        j++;

        /*
         * Fill in the chain element in the main frame.
         * About calculation on ChainOffset:
         * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
         *    in the end reserved for SGL element storage
         *    (MPI2_SGE_IO_UNION); we should count it in our
         *    calculation.  See its definition in the header file.
         * 2. Constant j is the counter of the current SGL element
         *    that will be processed, and (j - 1) is the number of
         *    SGL elements that have been processed (stored in the
         *    main frame).
         * 3. ChainOffset value should be in units of double-words (4
         *    bytes) so the last value should be divided by 4.
         */
        ddi_put8(acc_hdl, &frame->ChainOffset,
            (sizeof (MPI2_SCSI_IO_REQUEST) -
            sizeof (MPI2_SGE_IO_UNION) +
            (j - 1) * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
        sgechain = (pMpi2SGEChain64_t)sge;
        chainflags = (MPI2_SGE_FLAGS_CHAIN_ELEMENT |
            MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
            MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
        ddi_put8(acc_hdl, &sgechain->Flags, chainflags);

        /*
         * The size of the next frame is the accurate size of space
         * (in bytes) used to store the SGL elements. j is the counter
         * of SGL elements. (j - 1) is the number of SGL elements that
         * have been processed (stored in frames).
         */
        if (frames >= 2) {
                ASSERT(mpt->m_req_frame_size >= sizeof (MPI2_SGE_SIMPLE64));
                chainlength = mpt->m_req_frame_size /
                    sizeof (MPI2_SGE_SIMPLE64) *
                    sizeof (MPI2_SGE_SIMPLE64);
        } else {
                chainlength = ((cookiec - (j - 1)) *
                    sizeof (MPI2_SGE_SIMPLE64));
        }

        p = cmd->cmd_extra_frames;
```

```c
        ddi_put16(acc_hdl, &sgechain->Length, chainlength);
        ddi_put32(acc_hdl, &sgechain->Address.Low,
            p->m_phys_addr);
        /* SGL is allocated in the first 4G mem range */
        ddi_put32(acc_hdl, &sgechain->Address.High, 0);

        /*
         * If there are more than 2 frames left we have to
         * fill in the next chain offset to the location of
         * the chain element in the next frame.
         * sgemax is the number of simple elements in an extra
         * frame. Note that the value NextChainOffset should be
         * in double-words (4 bytes).
         */
        if (frames >= 2) {
                ddi_put8(acc_hdl, &sgechain->NextChainOffset,
                    (sgemax * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
        } else {
                ddi_put8(acc_hdl, &sgechain->NextChainOffset, 0);
        }

        /*
         * Jump to next frame;
         * Starting here, chain buffers go into the per command SGL.
         * This buffer is allocated when chain buffers are needed.
         */
        sge = (pMpi2SGESimple64_t)p->m_frames_addr;
        i = cookiec;

        /*
         * Start filling in frames with SGE's.  If we
         * reach the end of frame and still have SGE's
         * to fill we need to add a chain element and
         * use another frame.  j will be our counter
         * for what cookie we are at and i will be
         * the total cookiec. k is the current frame
         */
        for (k = 1; k <= frames; k++) {
                for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {

                        /*
                         * If we have reached the end of frame
                         * and we have more SGE's to fill in
                         * we have to fill the final entry
                         * with a chain element and then
                         * continue to the next frame
                         */
                        if ((l == (sgemax + 1)) && (k != frames)) {
                                sgechain = (pMpi2SGEChain64_t)sge;
                                j--;
                                chainflags = (
                                    MPI2_SGE_FLAGS_CHAIN_ELEMENT |
                                    MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
                                    MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
                                ddi_put8(p->m_acc_hdl,
                                    &sgechain->Flags, chainflags);
                                /*
                                 * k is the frame counter and (k + 1)
                                 * is the number of the next frame.
                                 * Note that frames are in contiguous
                                 * memory space.
                                 */
                                ddi_put32(p->m_acc_hdl,
                                    &sgechain->Address.Low,
                                    (p->m_phys_addr +
                                    (mpt->m_req_frame_size * k)));
                                ddi_put32(p->m_acc_hdl,
                                    &sgechain->Address.High, 0);

                                /*
                                 * If there are more than 2 frames left
                                 * we have to next chain offset to
                                 * the location of the chain element
                                 * in the next frame and fill in the
                                 * length of the next chain
                                 */
                                if ((frames - k) >= 2) {
                                        ddi_put8(p->m_acc_hdl,
                                            &sgechain->NextChainOffset,
                                            (sgemax *
                                            sizeof (MPI2_SGE_SIMPLE64))
                                            >> 2);
                                        ddi_put16(p->m_acc_hdl,
                                            &sgechain->Length,
                                            mpt->m_req_frame_size /
                                            sizeof (MPI2_SGE_SIMPLE64) *
                                            sizeof (MPI2_SGE_SIMPLE64));
                                } else {
                                        /*
                                         * This is the last frame. Set
                                         * the NextChainOffset to 0 and
                                         * Length is the total size of
                                         * all remaining simple elements
                                         */
                                        ddi_put8(p->m_acc_hdl,
                                            &sgechain->NextChainOffset,
                                            0);
                                        ddi_put16(p->m_acc_hdl,
                                            &sgechain->Length,
                                            (cookiec - j) *
                                            sizeof (MPI2_SGE_SIMPLE64));
                                }

                                /* Jump to the next frame */
                                sge = (pMpi2SGESimple64_t)
                                    ((char *)p->m_frames_addr +
                                    (int)mpt->m_req_frame_size * k);

                                continue;
                        }

                        ddi_put32(p->m_acc_hdl,
                            &sge->Address.Low,
                            dmap->addr.address64.Low);
                        ddi_put32(p->m_acc_hdl,
                            &sge->Address.High,
                            dmap->addr.address64.High);
                        ddi_put32(p->m_acc_hdl,
                            &sge->FlagsLength, dmap->count);
                        flags = ddi_get32(p->m_acc_hdl,
                            &sge->FlagsLength);
                        flags |= ((uint32_t)(
                            MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
                            MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
                            MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
                            MPI2_SGE_FLAGS_SHIFT);

                        /*
                         * If we are at the end of the frame and
                         * there is another frame to fill in
                         * we set the last simple element as last
```

```
                         * element
                         */
                        if ((l == sgemax) && (k != frames)) {
                                flags |= ((uint32_t)
                                    (MPI2_SGE_FLAGS_LAST_ELEMENT) <<
                                    MPI2_SGE_FLAGS_SHIFT);
                        }

                        /*
                         * If this is the final cookie we
                         * indicate it by setting the flags
                         */
                        if (j == i) {
                                flags |= ((uint32_t)
                                    (MPI2_SGE_FLAGS_LAST_ELEMENT |
                                    MPI2_SGE_FLAGS_END_OF_BUFFER |
                                    MPI2_SGE_FLAGS_END_OF_LIST) <<
                                    MPI2_SGE_FLAGS_SHIFT);
                        }
                        if (cmd->cmd_flags & CFLAG_DMASEND) {
                                flags |=
                                    (MPI2_SGE_FLAGS_HOST_TO_IOC <<
                                    MPI2_SGE_FLAGS_SHIFT);
                        } else {
                                flags |=
                                    (MPI2_SGE_FLAGS_IOC_TO_HOST <<
                                    MPI2_SGE_FLAGS_SHIFT);
                        }
                        ddi_put32(p->m_acc_hdl,
                            &sge->FlagsLength, flags);
                        dmap++;
                        sge++;
                }
        }

        /*
         * Sync DMA with the chain buffers that were just created
         */
        (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
}

static void
mptsas_ieee_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
    ddi_acc_handle_t acc_hdl, uint_t cookiec, uint8_t end_flag)
{
        pMpi2IeeeSgeSimple64_t  ieeesge;
        mptti_t                 *dmap;
        uint8_t                 flags;

        dmap = cmd->cmd_sg;

        NDBG1(("mptsas_ieee_sge_mainframe: cookiec=%d, %s", cookiec,
            cmd->cmd_flags & CFLAG_DMASEND?"Out":"In"));

        ieeesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);
        while (cookiec--) {
                ddi_put32(acc_hdl,
                    &ieeesge->Address.Low, dmap->addr.address64.Low);
                ddi_put32(acc_hdl,
                    &ieeesge->Address.High, dmap->addr.address64.High);
                ddi_put32(acc_hdl, &ieeesge->Length,
                    dmap->count);
                NDBG1(("mptsas_ieee_sge_mainframe: len=%d", dmap->count));
                flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
                    MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
                /*
                 * If this is the last cookie, we set the flags
                 * to indicate so
                 */
                if (cookiec == 0) {
                        flags |= end_flag;
                }

                ddi_put8(acc_hdl, &ieeesge->Flags, flags);
                dmap++;
                ieeesge++;
        }
}

static void
mptsas_ieee_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
    pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
{
        pMpi2IeeeSgeSimple64_t  ieeesge;
        pMpi25IeeeSgeChain64_t  ieeesgechain;
        uint_t                  cookiec;
        mptti_t                 *dmap;
        uint8_t                 flags;

        /*
         * Save the number of entries in the DMA
         * Scatter/Gather list
         */
        cookiec = cmd->cmd_cookiec;

        NDBG1(("mptsas_ieee_sge_chain: cookiec=%d", cookiec));

        /*
         * Hereby we start to deal with multiple frames.
         * The process is as follows:
         * 1. Determine how many frames are needed for SGL element
         *    storage; Note that all frames are stored in contiguous
         *    memory space and in 64-bit DMA mode each element is
         *    4 double-words (16 bytes) long.
         * 2. Fill up the main frame. We need to do this separately
         *    since it contains the SCSI IO request header and needs
         *    dedicated processing. Note that the last 4 double-words
         *    of the SCSI IO header is for SGL element storage
         *    (MPI2_SGE_IO_UNION).
         * 3. Fill the chain element in the main frame, so the DMA
         *    engine can use the following frames.
         * 4. Enter a loop to fill the remaining frames. Note that the
         *    last frame contains no chain element.  The remaining
         *    frames go into the mpt SGL buffer allocated on the fly,
         *    not immediately following the main message frame, as in
         *    Gen1.
         * Restrictions:
         *    For 64-bit DMA, the simple element and chain element
         *    are both of 4 double-words (16 bytes) in size, even
         *    though all frames are stored in the first 4G of mem
         *    range and the higher 32-bits of the address are always 0.
         */
        int                     i, j, k, l, frames, sgemax;
        int                     temp;
        uint8_t                 chainflags;
        uint32_t                chainlength;
        mptsas_cache_frames_t   *p;

        /*
         * Sgemax is the number of SGE's that will fit
         * each extra frame and frames is total
```

```
         * number of frames we'll need.  1 sge entry per
         * frame is reseverd for the chain element thus the -1 below.
         */
        sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_IEEE_SGE_SIMPLE64))
            - 1);
        temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;

        /*
         * A little check to see if we need to round up the number
         * of frames we need
         */
        if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
            sgemax) > 1) {
                frames = (temp + 1);
        } else {
                frames = temp;
        }
        NDBG1(("mptsas_ieee_sge_chain: temp=%d, frames=%d", temp, frames));
        dmap = cmd->cmd_sg;
        ieeesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);

        /*
         * First fill in the main frame
         */
        j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
        mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl, j, 0);
        dmap += j;
        ieeesge += j;
        j++;

        /*
         * Fill in the chain element in the main frame.
         * About calculation on ChainOffset:
         * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
         *    in the end reserved for SGL element storage
         *    (MPI2_SGE_IO_UNION); we should count it in our
         *    calculation.  See its definition in the header file.
         * 2. Constant j is the counter of the current SGL element
         *    that will be processed, and (j - 1) is the number of
         *    SGL elements that have been processed (stored in the
         *    main frame).
         * 3. ChainOffset value should be in units of quad-words (16
         *    bytes) so the last value should be divided by 16.
         */
        ddi_put8(acc_hdl, &frame->ChainOffset,
            (sizeof (MPI2_SCSI_IO_REQUEST) -
            sizeof (MPI2_SGE_IO_UNION) +
            (j - 1) * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
        ieeesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
        chainflags = (MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
            MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
        ddi_put8(acc_hdl, &ieeesgechain->Flags, chainflags);

        /*
         * The size of the next frame is the accurate size of space
         * (in bytes) used to store the SGL elements. j is the counter
         * of SGL elements. (j - 1) is the number of SGL elements that
         * have been processed (stored in frames).
         */
        if (frames >= 2) {
                ASSERT(mpt->m_req_frame_size >=
                    sizeof (MPI2_IEEE_SGE_SIMPLE64));
                chainlength = mpt->m_req_frame_size /
                    sizeof (MPI2_IEEE_SGE_SIMPLE64) *
                    sizeof (MPI2_IEEE_SGE_SIMPLE64);
        } else {
```

```
		chainlength = ((cookiec - (j - 1)) *
		    sizeof (MPI2_IEEE_SGE_SIMPLE64));
	}

	p = cmd->cmd_extra_frames;

	ddi_put32(acc_hdl, &ieeesgechain->Length, chainlength);
	ddi_put32(acc_hdl, &ieeesgechain->Address.Low,
	    p->m_phys_addr);
	/* SGL is allocated in the first 4G mem range */
	ddi_put32(acc_hdl, &ieeesgechain->Address.High, 0);

	/*
	 * If there are more than 2 frames left we have to
	 * fill in the next chain offset to the location of
	 * the chain element in the next frame.
	 * sgemax is the number of simple elements in an extra
	 * frame. Note that the value NextChainOffset should be
	 * in double-words (4 bytes).
	 */
	if (frames >= 2) {
		ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset,
		    (sgemax * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
	} else {
		ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset, 0);
	}

	/*
	 * Jump to next frame;
	 * Starting here, chain buffers go into the per command SGL.
	 * This buffer is allocated when chain buffers are needed.
	 */
	ieeesge = (pMpi2IeeeSgeSimple64_t)p->m_frames_addr;
	i = cookiec;

	/*
	 * Start filling in frames with SGE's.  If we
	 * reach the end of frame and still have SGE's
	 * to fill we need to add a chain element and
	 * use another frame.  j will be our counter
	 * for what cookie we are at and i will be
	 * the total cookiec. k is the current frame
	 */
	for (k = 1; k <= frames; k++) {
		for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {

			/*
			 * If we have reached the end of frame
			 * and we have more SGE's to fill in
			 * we have to fill the final entry
			 * with a chain element and then
			 * continue to the next frame
			 */
			if ((l == (sgemax + 1)) && (k != frames)) {
				ieeesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
				j--;
				chainflags =
				    MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
				    MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR;
				ddi_put8(p->m_acc_hdl,
				    &ieeesgechain->Flags, chainflags);
				/*
				 * k is the frame counter and (k + 1)
				 * is the number of the next frame.
				 * Note that frames are in contiguous
				 * memory space.
```

```
				 */
				ddi_put32(p->m_acc_hdl,
				    &ieeesgechain->Address.Low,
				    (p->m_phys_addr +
				    (mpt->m_req_frame_size * k)));
				ddi_put32(p->m_acc_hdl,
				    &ieeesgechain->Address.High, 0);

				/*
				 * If there are more than 2 frames left
				 * we have to next chain offset to
				 * the location of the chain element
				 * in the next frame and fill in the
				 * length of the next chain
				 */
				if ((frames - k) >= 2) {
					ddi_put8(p->m_acc_hdl,
					    &ieeesgechain->NextChainOffset,
					    (sgemax *
					    sizeof (MPI2_IEEE_SGE_SIMPLE64))
					    >> 4);
					ASSERT(mpt->m_req_frame_size >=
					    sizeof (MPI2_IEEE_SGE_SIMPLE64));
					ddi_put32(p->m_acc_hdl,
					    &ieeesgechain->Length,
					    mpt->m_req_frame_size /
					    sizeof (MPI2_IEEE_SGE_SIMPLE64) *
					    sizeof (MPI2_IEEE_SGE_SIMPLE64));
				} else {
					/*
					 * This is the last frame. Set
					 * the NextChainOffset to 0 and
					 * Length is the total size of
					 * all remaining simple elements
					 */
					ddi_put8(p->m_acc_hdl,
					    &ieeesgechain->NextChainOffset,
					    0);
					ddi_put32(p->m_acc_hdl,
					    &ieeesgechain->Length,
					    (cookiec - j) *
					    sizeof (MPI2_IEEE_SGE_SIMPLE64));
				}

				/* Jump to the next frame */
				ieeesge = (pMpi2IeeeSgeSimple64_t)
				    ((char *)p->m_frames_addr +
				    (int)mpt->m_req_frame_size * k);

				continue;
			}

			ddi_put32(p->m_acc_hdl,
			    &ieeesge->Address.Low,
			    dmap->addr.address64.Low);
			ddi_put32(p->m_acc_hdl,
			    &ieeesge->Address.High,
			    dmap->addr.address64.High);
			ddi_put32(p->m_acc_hdl,
			    &ieeesge->Length, dmap->count);
			flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
			    MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);

			/*
			 * If we are at the end of the frame and
			 * there is another frame to fill in
```

```
                         * do we need to do anything?
                         * if ((l == sgemax) && (k != frames)) {
                         * }
                         */

                        /*
                         * If this is the final cookie set end of list.
                         */
                        if (j == i) {
                                flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
                        }

                        ddi_put8(p->m_acc_hdl, &ieeesge->Flags, flags);
                        dmap++;
                        ieeesge++;
                }
        }

        /*
         * Sync DMA with the chain buffers that were just created
         */
        (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
}

static void
mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd, uint32_t *control,
    pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
{
        ASSERT(cmd->cmd_flags & CFLAG_DMAVALID);

        NDBG1(("mptsas_sge_setup: cookiec=%d", cmd->cmd_cookiec));

        /*
         * Set read/write bit in control.
         */
        if (cmd->cmd_flags & CFLAG_DMASEND) {
                *control |= MPI2_SCSIIO_CONTROL_WRITE;
        } else {
                *control |= MPI2_SCSIIO_CONTROL_READ;
        }

        ddi_put32(acc_hdl, &frame->DataLength, cmd->cmd_dmacount);

        /*
         * We have 4 cases here.  First where we can fit all the
         * SG elements into the main frame, and the case
         * where we can't. The SG element is also different when using
         * MPI2.5 interface.
         * If we have more cookies than we can attach to a frame
         * we will need to use a chain element to point
         * a location of memory where the rest of the S/G
         * elements reside.
         */
        if (cmd->cmd_cookiec <= MPTSAS_MAX_FRAME_SGES64(mpt)) {
                if (mpt->m_MPI25) {
                        mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl,
                            cmd->cmd_cookiec,
                            MPI25_IEEE_SGE_FLAGS_END_OF_LIST);
                } else {
                        mptsas_sge_mainframe(cmd, frame, acc_hdl,
                            cmd->cmd_cookiec,
                            ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT
                            | MPI2_SGE_FLAGS_END_OF_BUFFER
                            | MPI2_SGE_FLAGS_END_OF_LIST) <<
                            MPI2_SGE_FLAGS_SHIFT));
                }
```

```
        } else {
                if (mpt->m_MPI25) {
                        mptsas_ieee_sge_chain(mpt, cmd, frame, acc_hdl);
                } else {
                        mptsas_sge_chain(mpt, cmd, frame, acc_hdl);
                }
        }
}

/*
 * Interrupt handling
 * Utility routine.  Poll for status of a command sent to HBA
 * without interrupts (a FLAG_NOINTR command).
 */
int
mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
{
        int     rval = TRUE;

        NDBG5(("mptsas_poll: cmd=0x%p", (void *)poll_cmd));

        if ((poll_cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
                mptsas_restart_hba(mpt);
        }

        /*
         * Wait, using drv_usecwait(), long enough for the command to
         * reasonably return from the target if the target isn't
         * "dead".  A polled command may well be sent from scsi_poll, and
         * there are retries built in to scsi_poll if the transport
         * accepted the packet (TRAN_ACCEPT).  scsi_poll waits 1 second
         * and retries the transport up to scsi_poll_busycnt times
         * (currently 60) if
         * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
         * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
         *
         * limit the waiting to avoid a hang in the event that the
         * cmd never gets started but we are still receiving interrupts
         */
        while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
                if (mptsas_wait_intr(mpt, polltime) == FALSE) {
                        NDBG5(("mptsas_poll: command incomplete"));
                        rval = FALSE;
                        break;
                }
        }

        if (rval == FALSE) {

                /*
                 * this isn't supposed to happen, the hba must be wedged
                 * Mark this cmd as a timeout.
                 */
                mptsas_set_pkt_reason(mpt, poll_cmd, CMD_TIMEOUT,
                    (STAT_TIMEOUT|STAT_ABORTED));

                if (poll_cmd->cmd_queued == FALSE) {

                        NDBG5(("mptsas_poll: not on waitq"));

                        poll_cmd->cmd_pkt->pkt_state |=
                            (STATE_GOT_BUS|STATE_GOT_TARGET|STATE_SENT_CMD);
                } else {

                        /* find and remove it from the waitq */
                        NDBG5(("mptsas_poll: delete from waitq"));
                        mptsas_waitq_delete(mpt, poll_cmd);
                }

        }
        mptsas_fma_check(mpt, poll_cmd);
        NDBG5(("mptsas_poll: done"));
        return (rval);
}

/*
 * Used for polling cmds and TM function
 */
static int
mptsas_wait_intr(mptsas_t *mpt, int polltime)
{
        int                             cnt;
        pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
        uint32_t                        int_mask;

        NDBG5(("mptsas_wait_intr"));

        mpt->m_polled_intr = 1;

        /*
         * Get the current interrupt mask and disable interrupts.  When
         * re-enabling ints, set mask to saved value.
         */
        int_mask = ddi_get32(mpt->m_datap, &mpt->m_reg->HostInterruptMask);
        MPTSAS_DISABLE_INTR(mpt);

        /*
         * Keep polling for at least (polltime * 1000) seconds
         */
        for (cnt = 0; cnt < polltime; cnt++) {
                (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
                    DDI_DMA_SYNC_FORCPU);

                reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
                    MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

                if (ddi_get32(mpt->m_acc_post_queue_hdl,
                    &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
                    ddi_get32(mpt->m_acc_post_queue_hdl,
                    &reply_desc_union->Words.High) == 0xFFFFFFFF) {
                        drv_usecwait(1000);
                        continue;
                }

                /*
                 * The reply is valid, process it according to its
                 * type.
                 */
                mptsas_process_intr(mpt, reply_desc_union);

                if (++mpt->m_post_index == mpt->m_post_queue_depth) {
                        mpt->m_post_index = 0;
                }

                /*
                 * Update the global reply index
                 */
                ddi_put32(mpt->m_datap,
                    &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
                mpt->m_polled_intr = 0;

                /*
```

```
                 * Re-enable interrupts and quit.
                 */
                ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask,
                    int_mask);
                return (TRUE);

        }

        /*
         * Clear polling flag, re-enable interrupts and quit.
         */
        mpt->m_polled_intr = 0;
        ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, int_mask);
        return (FALSE);
}

static void
mptsas_handle_scsi_io_success(mptsas_t *mpt,
    pMpi2ReplyDescriptorsUnion_t reply_desc)
{
        pMpi2SCSIIOSuccessReplyDescriptor_t     scsi_io_success;
        uint16_t                                SMID;
        mptsas_slots_t                          *slots = mpt->m_active;
        mptsas_cmd_t                            *cmd = NULL;
        struct scsi_pkt                         *pkt;

        ASSERT(mutex_owned(&mpt->m_mutex));

        scsi_io_success = (pMpi2SCSIIOSuccessReplyDescriptor_t)reply_desc;
        SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &scsi_io_success->SMID);

        /*
         * This is a success reply so just complete the IO.  First, do a sanity
         * check on the SMID.  The final slot is used for TM requests, which
         * would not come into this reply handler.
         */
        if ((SMID == 0) || (SMID > slots->m_n_normal)) {
                mptsas_log(mpt, CE_WARN, "?Received invalid SMID of %d\n",
                    SMID);
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
                return;
        }

        cmd = slots->m_slot[SMID];

        /*
         * print warning and return if the slot is empty
         */
        if (cmd == NULL) {
                mptsas_log(mpt, CE_WARN, "?NULL command for successful SCSI IO "
                    "in slot %d", SMID);
                return;
        }

        pkt = CMD2PKT(cmd);
        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET | STATE_SENT_CMD |
            STATE_GOT_STATUS);
        if (cmd->cmd_flags & CFLAG_DMAVALID) {
                pkt->pkt_state |= STATE_XFERRED_DATA;
        }
        pkt->pkt_resid = 0;

        if (cmd->cmd_flags & CFLAG_PASSTHRU) {
                cmd->cmd_flags |= CFLAG_FINISHED;
                cv_broadcast(&mpt->m_passthru_cv);
                return;
        } else {
                mptsas_remove_cmd(mpt, cmd);
        }

        if (cmd->cmd_flags & CFLAG_RETRY) {
                /*
                 * The target returned QFULL or busy, do not add tihs
                 * pkt to the doneq since the hba will retry
                 * this cmd.
                 *
                 * The pkt has already been resubmitted in
                 * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
                 * Remove this cmd_flag here.
                 */
                cmd->cmd_flags &= ~CFLAG_RETRY;
        } else {
                mptsas_doneq_add(mpt, cmd);
        }
}

static void
mptsas_handle_address_reply(mptsas_t *mpt,
    pMpi2ReplyDescriptorsUnion_t reply_desc)
{
        pMpi2AddressReplyDescriptor_t   address_reply;
        pMPI2DefaultReply_t             reply;
        mptsas_fw_diagnostic_buffer_t   *pBuffer;
        uint32_t                        reply_addr;
        uint16_t                        SMID, iocstatus;
        mptsas_slots_t                  *slots = mpt->m_active;
        mptsas_cmd_t                    *cmd = NULL;
        uint8_t                         function, buffer_type;
        m_replyh_arg_t                  *args;
        int                             reply_frame_no;

        ASSERT(mutex_owned(&mpt->m_mutex));

        address_reply = (pMpi2AddressReplyDescriptor_t)reply_desc;
        reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
            &address_reply->ReplyFrameAddress);
        SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

        /*
         * If reply frame is not in the proper range we should ignore this
         * message and exit the interrupt handler.
         */
        if ((reply_addr < mpt->m_reply_frame_dma_addr) ||
            (reply_addr >= (mpt->m_reply_frame_dma_addr +
            (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
            ((reply_addr - mpt->m_reply_frame_dma_addr) %
            mpt->m_reply_frame_size != 0)) {
                mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
                    "address 0x%x\n", reply_addr);
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
                return;
        }

        (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
            DDI_DMA_SYNC_FORCPU);
        reply = (pMPI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
            mpt->m_reply_frame_dma_addr));
        function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

        NDBG31(("mptsas_handle_address_reply: function 0x%x, reply_addr=0x%x",
            function, reply_addr));
```

```
        /*
         * don't get slot information and command for events since these values
         * don't exist
         */
        if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
            (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
                /*
                 * This could be a TM reply, which use the last allocated SMID,
                 * so allow for that.
                 */
                if ((SMID == 0) || (SMID > (slots->m_n_normal + 1))) {
                        mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
                            "%d\n", SMID);
                        ddi_fm_service_impact(mpt->m_dip,
                            DDI_SERVICE_UNAFFECTED);
                        return;
                }

                cmd = slots->m_slot[SMID];

                /*
                 * print warning and return if the slot is empty
                 */
                if (cmd == NULL) {
                        mptsas_log(mpt, CE_WARN, "?NULL command for address "
                            "reply in slot %d", SMID);
                        return;
                }
                if ((cmd->cmd_flags &
                    (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG))) {
                        cmd->cmd_rfm = reply_addr;
                        cmd->cmd_flags |= CFLAG_FINISHED;
                        cv_broadcast(&mpt->m_passthru_cv);
                        cv_broadcast(&mpt->m_config_cv);
                        cv_broadcast(&mpt->m_fw_diag_cv);
                        return;
                } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
                        mptsas_remove_cmd(mpt, cmd);
                }
                NDBG31(("\t\tmptsas_process_intr: slot=%d", SMID));
        }
        /*
         * Depending on the function, we need to handle
         * the reply frame (and cmd) differently.
         */
        switch (function) {
        case MPI2_FUNCTION_SCSI_IO_REQUEST:
                mptsas_check_scsi_io_error(mpt, (pMpi2SCSIIOReply_t)reply, cmd);
                break;
        case MPI2_FUNCTION_SCSI_TASK_MGMT:
                cmd->cmd_rfm = reply_addr;
                mptsas_check_task_mgt(mpt, (pMpi2SCSIManagementReply_t)reply,
                    cmd);
                break;
        case MPI2_FUNCTION_FW_DOWNLOAD:
                cmd->cmd_flags |= CFLAG_FINISHED;
                cv_signal(&mpt->m_fw_cv);
                break;
        case MPI2_FUNCTION_EVENT_NOTIFICATION:
                reply_frame_no = (reply_addr - mpt->m_reply_frame_dma_addr) /
                    mpt->m_reply_frame_size;
                args = &mpt->m_replyh_args[reply_frame_no];
                args->mpt = (void *)mpt;
                args->rfm = reply_addr;

                /*
```

```
                 * Record the event if its type is enabled in
                 * this mpt instance by ioctl.
                 */
                mptsas_record_event(args);

                /*
                 * Handle time critical events
                 * NOT_RESPONDING/ADDED only now
                 */
                if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {
                        /*
                         * Would not return main process,
                         * just let taskq resolve ack action
                         * and ack would be sent in taskq thread
                         */
                        NDBG20(("send mptsas_handle_event_sync success"));
                }

                if (mpt->m_in_reset) {
                        NDBG20(("dropping event received during reset"));
                        return;
                }

                if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
                    (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
                        mptsas_log(mpt, CE_WARN, "No memory available"
                        "for dispatch taskq");
                        /*
                         * Return the reply frame to the free queue.
                         */
                        ddi_put32(mpt->m_acc_free_queue_hdl,
                            &((uint32_t *)(void *)
                            mpt->m_free_queue)[mpt->m_free_index], reply_addr);
                        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
                            DDI_DMA_SYNC_FORDEV);
                        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                                mpt->m_free_index = 0;
                        }

                        ddi_put32(mpt->m_datap,
                            &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
                }
                return;
        case MPI2_FUNCTION_DIAG_BUFFER_POST:
                /*
                 * If SMID is 0, this implies that the reply is due to a
                 * release function with a status that the buffer has been
                 * released.  Set the buffer flags accordingly.
                 */
                if (SMID == 0) {
                        iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
                            &reply->IOCStatus);
                        buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
                            &(((pMpi2DiagBufferPostReply_t)reply)->BufferType));
                        if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
                                pBuffer =
                                    &mpt->m_fw_diag_buffer_list[buffer_type];
                                pBuffer->valid_data = TRUE;
                                pBuffer->owned_by_firmware = FALSE;
                                pBuffer->immediate = FALSE;
                        }
                } else {
                        /*
                         * Normal handling of diag post reply with SMID.
                         */
                        cmd = slots->m_slot[SMID];
```

```

                        /*
                         * print warning and return if the slot is empty
                         */
                        if (cmd == NULL) {
                                mptsas_log(mpt, CE_WARN, "?NULL command for "
                                    "address reply in slot %d", SMID);
                                return;
                        }
                        cmd->cmd_rfm = reply_addr;
                        cmd->cmd_flags |= CFLAG_FINISHED;
                        cv_broadcast(&mpt->m_fw_diag_cv);
                }
                return;
        default:
                mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
                break;
        }

        /*
         * Return the reply frame to the free queue.
         */
        ddi_put32(mpt->m_acc_free_queue_hdl,
            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
            reply_addr);
        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);
        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                mpt->m_free_index = 0;
        }
        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
            mpt->m_free_index);

        if (cmd->cmd_flags & CFLAG_FW_CMD)
                return;

        if (cmd->cmd_flags & CFLAG_RETRY) {
                /*
                 * The target returned QFULL or busy, do not add this
                 * pkt to the doneq since the hba will retry
                 * this cmd.
                 *
                 * The pkt has already been resubmitted in
                 * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
                 * Remove this cmd_flag here.
                 */
                cmd->cmd_flags &= ~CFLAG_RETRY;
        } else {
                mptsas_doneq_add(mpt, cmd);
        }
}

static void
mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
    mptsas_cmd_t *cmd)
{
        uint8_t                 scsi_status, scsi_state;
        uint16_t                ioc_status;
        uint32_t                xferred, sensecount, responsedata, loginfo = 0;
        struct scsi_pkt         *pkt;
        struct scsi_arq_status  *arqstat;
        struct buf              *bp;
        mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
        uint8_t                 *sensedata = NULL;
        uint64_t                sas_wwn;
        uint8_t                 phy;
```

```
	char			wwn_str[MPTSAS_WWN_STRLEN];

	if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
	    (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
		bp = cmd->cmd_ext_arq_buf;
	} else {
		bp = cmd->cmd_arq_buf;
	}

	scsi_status = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIStatus);
	ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
	scsi_state = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIState);
	xferred = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->TransferCount);
	sensecount = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->SenseCount);
	responsedata = ddi_get32(mpt->m_acc_reply_frame_hdl,
	    &reply->ResponseInfo);

	if (ioc_status & MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
		sas_wwn = ptgt->m_addr.mta_wwn;
		phy = ptgt->m_phynum;
		if (sas_wwn == 0) {
			(void) sprintf(wwn_str, "p%x", phy);
		} else {
			(void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
		}
		loginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    &reply->IOCLogInfo);
		mptsas_log(mpt, CE_NOTE,
		    "?Log info 0x%x received for target %d %s.\n"
		    "\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
		    loginfo, Tgt(cmd), wwn_str, scsi_status, ioc_status,
		    scsi_state);
	}

	NDBG31(("\t\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
	    scsi_status, ioc_status, scsi_state));

	pkt = CMD2PKT(cmd);
	*(pkt->pkt_scbp) = scsi_status;

	if (loginfo == 0x31170000) {
		/*
		 * if loginfo PL_LOGINFO_CODE_IO_DEVICE_MISSING_DELAY_RETRY
		 * 0x31170000 comes, that means the device missing delay
		 * is in progressing, the command need retry later.
		 */
		*(pkt->pkt_scbp) = STATUS_BUSY;
		return;
	}

	if ((scsi_state & MPI2_SCSI_STATE_NO_SCSI_STATUS) &&
	    ((ioc_status & MPI2_IOCSTATUS_MASK) ==
	    MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE)) {
		pkt->pkt_reason = CMD_INCOMPLETE;
		pkt->pkt_state |= STATE_GOT_BUS;
		if (ptgt->m_reset_delay == 0) {
			mptsas_set_throttle(mpt, ptgt,
			    DRAIN_THROTTLE);
		}
		return;
	}

	if (scsi_state & MPI2_SCSI_STATE_RESPONSE_INFO_VALID) {
		responsedata &= 0x000000FF;
		if (responsedata & MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF) {
			mptsas_log(mpt, CE_NOTE, "Do not support the TLR\n");
```

```
			pkt->pkt_reason = CMD_TLR_OFF;
			return;
		}
	}


	switch (scsi_status) {
	case MPI2_SCSI_STATUS_CHECK_CONDITION:
		pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
		arqstat = (void*)(pkt->pkt_scbp);
		arqstat->sts_rqpkt_status = *((struct scsi_status *)
		    (pkt->pkt_scbp));
		pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET |
		    STATE_SENT_CMD | STATE_GOT_STATUS | STATE_ARQ_DONE);
		if (cmd->cmd_flags & CFLAG_XARQ) {
			pkt->pkt_state |= STATE_XARQ_DONE;
		}
		if (pkt->pkt_resid != cmd->cmd_dmacount) {
			pkt->pkt_state |= STATE_XFERRED_DATA;
		}
		arqstat->sts_rqpkt_reason = pkt->pkt_reason;
		arqstat->sts_rqpkt_state  = pkt->pkt_state;
		arqstat->sts_rqpkt_state |= STATE_XFERRED_DATA;
		arqstat->sts_rqpkt_statistics = pkt->pkt_statistics;
		sensedata = (uint8_t *)&arqstat->sts_sensedata;

		bcopy((uchar_t *)bp->b_un.b_addr, sensedata,
		    ((cmd->cmd_rqslen >= sensecount) ? sensecount :
		    cmd->cmd_rqslen));
		arqstat->sts_rqpkt_resid = (cmd->cmd_rqslen - sensecount);
		cmd->cmd_flags |= CFLAG_CMDARQ;
		/*
		 * Set proper status for pkt if autosense was valid
		 */
		if (scsi_state & MPI2_SCSI_STATE_AUTOSENSE_VALID) {
			struct scsi_status zero_status = { 0 };
			arqstat->sts_rqpkt_status = zero_status;
		}

		/*
		 * ASC=0x47 is parity error
		 * ASC=0x48 is initiator detected error received
		 */
		if ((scsi_sense_key(sensedata) == KEY_ABORTED_COMMAND) &&
		    ((scsi_sense_asc(sensedata) == 0x47) ||
		    (scsi_sense_asc(sensedata) == 0x48))) {
			mptsas_log(mpt, CE_NOTE, "Aborted_command!");
		}

		/*
		 * ASC/ASCQ=0x3F/0x0E means report_luns data changed
		 * ASC/ASCQ=0x25/0x00 means invalid lun
		 */
		if (((scsi_sense_key(sensedata) == KEY_UNIT_ATTENTION) &&
		    (scsi_sense_asc(sensedata) == 0x3F) &&
		    (scsi_sense_ascq(sensedata) == 0x0E)) ||
		    ((scsi_sense_key(sensedata) == KEY_ILLEGAL_REQUEST) &&
		    (scsi_sense_asc(sensedata) == 0x25) &&
		    (scsi_sense_ascq(sensedata) == 0x00))) {
			mptsas_topo_change_list_t *topo_node = NULL;

			topo_node = kmem_zalloc(
			    sizeof (mptsas_topo_change_list_t),
			    KM_NOSLEEP);
			if (topo_node == NULL) {
				mptsas_log(mpt, CE_NOTE, "No memory"
                                    "resource for handle SAS dynamic"
                                    "reconfigure.\n");
                                break;
                        }
                        topo_node->mpt = mpt;
                        topo_node->event = MPTSAS_DR_EVENT_RECONFIG_TARGET;
                        topo_node->un.phymask = ptgt->m_addr.mta_phymask;
                        topo_node->devhdl = ptgt->m_devhdl;
                        topo_node->object = (void *)ptgt;
                        topo_node->flags = MPTSAS_TOPO_FLAG_LUN_ASSOCIATED;

                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
                            mptsas_handle_dr,
                            (void *)topo_node,
                            DDI_NOSLEEP)) != DDI_SUCCESS) {
                                kmem_free(topo_node,
                                    sizeof (mptsas_topo_change_list_t));
                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq"
                                    "for handle SAS dynamic reconfigure"
                                    "failed. \n");
                        }
                }
                break;
        case MPI2_SCSI_STATUS_GOOD:
                switch (ioc_status & MPI2_IOCSTATUS_MASK) {
                case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
                        pkt->pkt_reason = CMD_DEV_GONE;
                        pkt->pkt_state |= STATE_GOT_BUS;
                        if (ptgt->m_reset_delay == 0) {
                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
                        }
                        NDBG31(("lost disk for target%d, command:%x",
                            Tgt(cmd), pkt->pkt_cdbp[0]));
                        break;
                case MPI2_IOCSTATUS_SCSI_DATA_OVERRUN:
                        NDBG31(("data overrun: xferred=%d", xferred));
                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
                        pkt->pkt_reason = CMD_DATA_OVR;
                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
                            | STATE_SENT_CMD | STATE_GOT_STATUS
                            | STATE_XFERRED_DATA);
                        pkt->pkt_resid = 0;
                        break;
                case MPI2_IOCSTATUS_SCSI_RESIDUAL_MISMATCH:
                case MPI2_IOCSTATUS_SCSI_DATA_UNDERRUN:
                        NDBG31(("data underrun: xferred=%d", xferred));
                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
                            | STATE_SENT_CMD | STATE_GOT_STATUS);
                        pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
                        if (pkt->pkt_resid != cmd->cmd_dmacount) {
                                pkt->pkt_state |= STATE_XFERRED_DATA;
                        }
                        break;
                case MPI2_IOCSTATUS_SCSI_TASK_TERMINATED:
                        if (cmd->cmd_active_expiration <= gethrtime()) {
                                /*
                                 * When timeout requested, propagate
                                 * proper reason and statistics to
                                 * target drivers.
                                 */
                                mptsas_set_pkt_reason(mpt, cmd, CMD_TIMEOUT,
                                    STAT_BUS_RESET | STAT_TIMEOUT);
                        } else {
                                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
                                    STAT_BUS_RESET);
```

```
                        }
                        break;
                case MPI2_IOCSTATUS_SCSI_IOC_TERMINATED:
                case MPI2_IOCSTATUS_SCSI_EXT_TERMINATED:
                        mptsas_set_pkt_reason(mpt,
                            cmd, CMD_RESET, STAT_DEV_RESET);
                        break;
                case MPI2_IOCSTATUS_SCSI_IO_DATA_ERROR:
                case MPI2_IOCSTATUS_SCSI_PROTOCOL_ERROR:
                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET);
                        mptsas_set_pkt_reason(mpt,
                            cmd, CMD_TERMINATED, STAT_TERMINATED);
                        break;
                case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
                case MPI2_IOCSTATUS_BUSY:
                        /*
                         * set throttles to drain
                         */
                        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
                            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
                        }

                        /*
                         * retry command
                         */
                        cmd->cmd_flags |= CFLAG_RETRY;
                        cmd->cmd_pkt_flags |= FLAG_HEAD;

                        (void) mptsas_accept_pkt(mpt, cmd);
                        break;
                default:
                        mptsas_log(mpt, CE_WARN,
                            "unknown ioc_status = %x\n", ioc_status);
                        mptsas_log(mpt, CE_CONT, "scsi_state = %x, transfer "
                            "count = %x, scsi_status = %x", scsi_state,
                            xferred, scsi_status);
                        break;
                }
                break;
        case MPI2_SCSI_STATUS_TASK_SET_FULL:
                mptsas_handle_qfull(mpt, cmd);
                break;
        case MPI2_SCSI_STATUS_BUSY:
                NDBG31(("scsi_status busy received"));
                break;
        case MPI2_SCSI_STATUS_RESERVATION_CONFLICT:
                NDBG31(("scsi_status reservation conflict received"));
                break;
        default:
                mptsas_log(mpt, CE_WARN, "scsi_status=%x, ioc_status=%x\n",
                    scsi_status, ioc_status);
                mptsas_log(mpt, CE_WARN,
                    "mptsas_process_intr: invalid scsi status\n");
                break;
        }
}

static void
mptsas_check_task_mgt(mptsas_t *mpt, pMpi2SCSIManagementReply_t reply,
        mptsas_cmd_t *cmd)
{
        uint8_t         task_type;
        uint16_t        ioc_status;
        uint32_t        log_info;
        uint16_t        dev_handle;
        struct scsi_pkt *pkt = CMD2PKT(cmd);

        task_type = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->TaskType);
        ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
        log_info = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->IOCLogInfo);
        dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->DevHandle);

        if (ioc_status != MPI2_IOCSTATUS_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "mptsas_check_task_mgt: Task 0x%x "
                    "failed. IOCStatus=0x%x IOCLogInfo=0x%x target=%d\n",
                    task_type, ioc_status, log_info, dev_handle);
                pkt->pkt_reason = CMD_INCOMPLETE;
                return;
        }

        switch (task_type) {
        case MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK:
        case MPI2_SCSITASKMGMT_TASKTYPE_CLEAR_TASK_SET:
        case MPI2_SCSITASKMGMT_TASKTYPE_QUERY_TASK:
        case MPI2_SCSITASKMGMT_TASKTYPE_CLR_ACA:
        case MPI2_SCSITASKMGMT_TASKTYPE_QRY_TASK_SET:
        case MPI2_SCSITASKMGMT_TASKTYPE_QRY_UNIT_ATTENTION:
                break;
        case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
        case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
        case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
                /*
                 * Check for invalid DevHandle of 0 in case application
                 * sends bad command.  DevHandle of 0 could cause problems.
                 */
                if (dev_handle == 0) {
                        mptsas_log(mpt, CE_WARN, "!Can't flush target with"
                            " DevHandle of 0.");
                } else {
                        mptsas_flush_target(mpt, dev_handle, Lun(cmd),
                            task_type);
                }
                break;
        default:
                mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
                    task_type);
                mptsas_log(mpt, CE_WARN, "ioc status = %x", ioc_status);
                break;
        }
}

static void
mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg)
{
        mptsas_t                        *mpt = arg->mpt;
        uint64_t                        t = arg->t;
        mptsas_cmd_t                    *cmd;
        struct scsi_pkt                 *pkt;
        mptsas_doneq_thread_list_t      *item = &mpt->m_doneq_thread_id[t];

        mutex_enter(&item->mutex);
        while (item->flag & MPTSAS_DONEQ_THREAD_ACTIVE) {
                if (!item->doneq) {
                        cv_wait(&item->cv, &item->mutex);
                }
                pkt = NULL;
                if ((cmd = mptsas_doneq_thread_rm(mpt, t)) != NULL) {
                        cmd->cmd_flags |= CFLAG_COMPLETED;
                        pkt = CMD2PKT(cmd);
                }
                mutex_exit(&item->mutex);
```

```
                if (pkt) {
                        mptsas_pkt_comp(pkt, cmd);
                }
                mutex_enter(&item->mutex);
        }
        mutex_exit(&item->mutex);
        mutex_enter(&mpt->m_doneq_mutex);
        mpt->m_doneq_thread_n--;
        cv_broadcast(&mpt->m_doneq_thread_cv);
        mutex_exit(&mpt->m_doneq_mutex);
}


/*
 * mpt interrupt handler.
 */
static uint_t
mptsas_intr(caddr_t arg1, caddr_t arg2)
{
        mptsas_t                        *mpt = (void *)arg1;
        pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
        uchar_t                         did_reply = FALSE;

        NDBG1(("mptsas_intr: arg1 0x%p arg2 0x%p", (void *)arg1, (void *)arg2));

        mutex_enter(&mpt->m_mutex);

        /*
         * If interrupts are shared by two channels then check whether this
         * interrupt is genuinely for this channel by making sure first the
         * chip is in high power state.
         */
        if ((mpt->m_options & MPTSAS_OPT_PM) &&
            (mpt->m_power_level != PM_LEVEL_D0)) {
                mutex_exit(&mpt->m_mutex);
                return (DDI_INTR_UNCLAIMED);
        }

        /*
         * If polling, interrupt was triggered by some shared interrupt because
         * IOC interrupts are disabled during polling, so polling routine will
         * handle any replies.  Considering this, if polling is happening,
         * return with interrupt unclaimed.
         */
        if (mpt->m_polled_intr) {
                mutex_exit(&mpt->m_mutex);
                mptsas_log(mpt, CE_WARN, "mpt_sas: Unclaimed interrupt");
                return (DDI_INTR_UNCLAIMED);
        }

        /*
         * Read the istat register.
         */
        if ((INTPENDING(mpt)) != 0) {
                /*
                 * read fifo until empty.
                 */
#ifndef __lock_lint
                _NOTE(CONSTCOND)
#endif
                while (TRUE) {
                        (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
                            DDI_DMA_SYNC_FORCPU);
                        reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
                            MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);
                        if (ddi_get32(mpt->m_acc_post_queue_hdl,
                            &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
                            ddi_get32(mpt->m_acc_post_queue_hdl,
                            &reply_desc_union->Words.High) == 0xFFFFFFFF) {
                                break;
                        }

                        /*
                         * The reply is valid, process it according to its
                         * type.  Also, set a flag for updating the reply index
                         * after they've all been processed.
                         */
                        did_reply = TRUE;

                        mptsas_process_intr(mpt, reply_desc_union);

                        /*
                         * Increment post index and roll over if needed.
                         */
                        if (++mpt->m_post_index == mpt->m_post_queue_depth) {
                                mpt->m_post_index = 0;
                        }
                }

                /*
                 * Update the global reply index if at least one reply was
                 * processed.
                 */
                if (did_reply) {
                        ddi_put32(mpt->m_datap,
                            &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
                }
        } else {
                mutex_exit(&mpt->m_mutex);
                return (DDI_INTR_UNCLAIMED);
        }
        NDBG1(("mptsas_intr complete"));

        /*
         * If no helper threads are created, process the doneq in ISR. If
         * helpers are created, use the doneq length as a metric to measure the
         * load on the interrupt CPU. If it is long enough, which indicates the
         * load is heavy, then we deliver the IO completions to the helpers.
         * This measurement has some limitations, although it is simple and
         * straightforward and works well for most of the cases at present.
         */
        if (!mpt->m_doneq_thread_n ||
            (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)) {
                mptsas_doneq_empty(mpt);
        } else {
                mptsas_deliver_doneq_thread(mpt);
        }

        /*
         * If there are queued cmd, start them now.
         */
        if (mpt->m_waitq != NULL) {
                mptsas_restart_waitq(mpt);
        }

        mutex_exit(&mpt->m_mutex);
        return (DDI_INTR_CLAIMED);
}

static void
mptsas_process_intr(mptsas_t *mpt,
```

```
    pMpi2ReplyDescriptorsUnion_t reply_desc_union)
{
        uint8_t reply_type;

        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * The reply is valid, process it according to its
         * type.  Also, set a flag for updated the reply index
         * after they've all been processed.
         */
        reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
            &reply_desc_union->Default.ReplyFlags);
        reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
        if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS ||
            reply_type == MPI25_RPY_DESCRIPT_FLAGS_FAST_PATH_SCSI_IO_SUCCESS) {
                mptsas_handle_scsi_io_success(mpt, reply_desc_union);
        } else if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
                mptsas_handle_address_reply(mpt, reply_desc_union);
        } else {
                mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
        }

        /*
         * Clear the reply descriptor for re-use and increment
         * index.
         */
        ddi_put64(mpt->m_acc_post_queue_hdl,
            &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
            0xFFFFFFFFFFFFFFFF);
        (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);
}

/*
 * handle qfull condition
 */
static void
mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

        if ((++cmd->cmd_qfull_retries > ptgt->m_qfull_retries) ||
            (ptgt->m_qfull_retries == 0)) {
                /*
                 * We have exhausted the retries on QFULL, or,
                 * the target driver has indicated that it
                 * wants to handle QFULL itself by setting
                 * qfull-retries capability to 0. In either case
                 * we want the target driver's QFULL handling
                 * to kick in. We do this by having pkt_reason
                 * as CMD_CMPLT and pkt_scbp as STATUS_QFULL.
                 */
                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
        } else {
                if (ptgt->m_reset_delay == 0) {
                        ptgt->m_t_throttle =
                            max((ptgt->m_t_ncmds - 2), 0);
                }

                cmd->cmd_pkt_flags |= FLAG_HEAD;
                cmd->cmd_flags &= ~(CFLAG_TRANFLAG);
                cmd->cmd_flags |= CFLAG_RETRY;

                (void) mptsas_accept_pkt(mpt, cmd);
                /*
                 * when target gives queue full status with no commands
                 * outstanding (m_t_ncmds == 0), throttle is set to 0
                 * (HOLD_THROTTLE), and the queue full handling start
                 * (see psarc/1994/313); if there are commands outstanding,
                 * throttle is set to (m_t_ncmds - 2)
                 */
                if (ptgt->m_t_throttle == HOLD_THROTTLE) {
                        /*
                         * By setting throttle to QFULL_THROTTLE, we
                         * avoid submitting new commands and in
                         * mptsas_restart_cmd find out slots which need
                         * their throttles to be cleared.
                         */
                        mptsas_set_throttle(mpt, ptgt, QFULL_THROTTLE);
                        if (mpt->m_restart_cmd_timeid == 0) {
                                mpt->m_restart_cmd_timeid =
                                    timeout(mptsas_restart_cmd, mpt,
                                    ptgt->m_qfull_retry_interval);
                        }
                }
        }
}

mptsas_phymask_t
mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport)
{
        mptsas_phymask_t        phy_mask = 0;
        uint8_t                 i = 0;

        NDBG20(("mptsas%d physport_to_phymask enter", mpt->m_instance));

        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * If physport is 0xFF, this is a RAID volume.  Use phymask of 0.
         */
        if (physport == 0xFF) {
                return (0);
        }

        for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
                if (mpt->m_phy_info[i].attached_devhdl &&
                    (mpt->m_phy_info[i].phy_mask != 0) &&
                    (mpt->m_phy_info[i].port_num == physport)) {
                        phy_mask = mpt->m_phy_info[i].phy_mask;
                        break;
                }
        }
        NDBG20(("mptsas%d physport_to_phymask:physport :%x phymask :%x, ",
            mpt->m_instance, physport, phy_mask));
        return (phy_mask);
}

/*
 * mpt free device handle after device gone, by use of passthrough
 */
static int
mptsas_free_devhdl(mptsas_t *mpt, uint16_t devhdl)
{
        Mpi2SasIoUnitControlRequest_t   req;
        Mpi2SasIoUnitControlReply_t     rep;
        int                             ret;

        ASSERT(mutex_owned(&mpt->m_mutex));
```

```
        /*
         * Need to compose a SAS IO Unit Control request message
         * and call mptsas_do_passthru() function
         */
        bzero(&req, sizeof (req));
        bzero(&rep, sizeof (rep));

        req.Function = MPI2_FUNCTION_SAS_IO_UNIT_CONTROL;
        req.Operation = MPI2_SAS_OP_REMOVE_DEVICE;
        req.DevHandle = LE_16(devhdl);

        ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
            sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
        if (ret != 0) {
                cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
                    "Control error %d", ret);
                return (DDI_FAILURE);
        }

        /* do passthrough success, check the ioc status */
        if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
                cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
                    "Control IOCStatus %d", LE_16(rep.IOCStatus));
                return (DDI_FAILURE);
        }

        return (DDI_SUCCESS);
}

static void
mptsas_update_phymask(mptsas_t *mpt)
{
        mptsas_phymask_t mask = 0, phy_mask;
        char            *phy_mask_name;
        uint8_t         current_port;
        int             i, j;

        NDBG20(("mptsas%d update phymask ", mpt->m_instance));

        ASSERT(mutex_owned(&mpt->m_mutex));

        (void) mptsas_get_sas_io_unit_page(mpt);

        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);

        for (i = 0; i < mpt->m_num_phys; i++) {
                phy_mask = 0x00;

                if (mpt->m_phy_info[i].attached_devhdl == 0)
                        continue;

                bzero(phy_mask_name, sizeof (phy_mask_name));

                current_port = mpt->m_phy_info[i].port_num;

                if ((mask & (1 << i)) != 0)
                        continue;

                for (j = 0; j < mpt->m_num_phys; j++) {
                        if (mpt->m_phy_info[j].attached_devhdl &&
                            (mpt->m_phy_info[j].port_num == current_port)) {
                                phy_mask |= (1 << j);
                        }
                }
                mask = mask | phy_mask;
```

```c
                for (j = 0; j < mpt->m_num_phys; j++) {
                        if ((phy_mask >> j) & 0x01) {
                                mpt->m_phy_info[j].phy_mask = phy_mask;
                        }
                }

                (void) sprintf(phy_mask_name, "%x", phy_mask);

                mutex_exit(&mpt->m_mutex);
                /*
                 * register a iport, if the port has already been existed
                 * SCSA will do nothing and just return.
                 */
                (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
                mutex_enter(&mpt->m_mutex);
        }
        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
        NDBG20(("mptsas%d update phymask return", mpt->m_instance));
}

/*
 * mptsas_handle_dr is a task handler for DR, the DR action includes:
 * 1. Directly attched Device Added/Removed.
 * 2. Expander Device Added/Removed.
 * 3. Indirectly Attached Device Added/Expander.
 * 4. LUNs of a existing device status change.
 * 5. RAID volume created/deleted.
 * 6. Member of RAID volume is released because of RAID deletion.
 * 7. Physical disks are removed because of RAID creation.
 */
static void
mptsas_handle_dr(void *args) {
        mptsas_topo_change_list_t       *topo_node = NULL;
        mptsas_topo_change_list_t       *save_node = NULL;
        mptsas_t                        *mpt;
        dev_info_t                      *parent = NULL;
        mptsas_phymask_t                phymask = 0;
        char                            *phy_mask_name;
        uint8_t                         flags = 0, physport = 0xff;
        uint8_t                         port_update = 0;
        uint_t                          event;

        topo_node = (mptsas_topo_change_list_t *)args;

        mpt = topo_node->mpt;
        event = topo_node->event;
        flags = topo_node->flags;

        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);

        NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));

        switch (event) {
        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
                        /*
                         * Direct attached or expander attached device added
                         * into system or a Phys Disk that is being unhidden.
                         */
                        port_update = 1;
                }
                break;
        case MPTSAS_DR_EVENT_RECONFIG_SMP:
```

```c
                /*
                 * New expander added into system, it must be the head
                 * of topo_change_list_t
                 */
                port_update = 1;
                break;
        default:
                port_update = 0;
                break;
        }
        /*
         * All cases port_update == 1 may cause initiator port form change
         */
        mutex_enter(&mpt->m_mutex);
        if (mpt->m_port_chng && port_update) {
                /*
                 * mpt->m_port_chng flag indicates some PHYs of initiator
                 * port have changed to online. So when expander added or
                 * directly attached device online event come, we force to
                 * update port information by issueing SAS IO Unit Page and
                 * update PHYMASKs.
                 */
                (void) mptsas_update_phymask(mpt);
                mpt->m_port_chng = 0;

        }
        mutex_exit(&mpt->m_mutex);
        while (topo_node) {
                phymask = 0;
                if (parent == NULL) {
                        physport = topo_node->un.physport;
                        event = topo_node->event;
                        flags = topo_node->flags;
                        if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
                            MPTSAS_DR_EVENT_OFFLINE_SMP)) {
                                /*
                                 * For all offline events, phymask is known
                                 */
                                phymask = topo_node->un.phymask;
                                goto find_parent;
                        }
                        if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
                                goto handle_topo_change;
                        }
                        if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
                                phymask = topo_node->un.phymask;
                                goto find_parent;
                        }

                        if ((flags ==
                            MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
                            (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
                                /*
                                 * There is no any field in IR_CONFIG_CHANGE
                                 * event indicate physport/phynum, let's get
                                 * parent after SAS Device Page0 request.
                                 */
                                goto handle_topo_change;
                        }

                        mutex_enter(&mpt->m_mutex);
                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
                                /*
                                 * If the direct attached device added or a
                                 * phys disk is being unhidden, argument
                                 * physport actually is PHY#, so we have to get
                                 * phymask according PHY#.
                                 */
                                physport = mpt->m_phy_info[physport].port_num;
                        }

                        /*
                         * Translate physport to phymask so that we can search
                         * parent dip.
                         */
                        phymask = mptsas_physport_to_phymask(mpt,
                            physport);
                        mutex_exit(&mpt->m_mutex);

find_parent:
                        bzero(phy_mask_name, MPTSAS_MAX_PHYS);
                        /*
                         * For RAID topology change node, write the iport name
                         * as v0.
                         */
                        if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
                                (void) sprintf(phy_mask_name, "v0");
                        } else {
                                /*
                                 * phymask can bo 0 if the drive has been
                                 * pulled by the time an add event is
                                 * processed.  If phymask is 0, just skip this
                                 * event and continue.
                                 */
                                if (phymask == 0) {
                                        mutex_enter(&mpt->m_mutex);
                                        save_node = topo_node;
                                        topo_node = topo_node->next;
                                        ASSERT(save_node);
                                        kmem_free(save_node,
                                            sizeof (mptsas_topo_change_list_t));
                                        mutex_exit(&mpt->m_mutex);

                                        parent = NULL;
                                        continue;
                                }
                                (void) sprintf(phy_mask_name, "%x", phymask);
                        }
                        parent = scsi_hba_iport_find(mpt->m_dip,
                            phy_mask_name);
                        if (parent == NULL) {
                                mptsas_log(mpt, CE_WARN, "Failed to find an "
                                    "iport, should not happen!");
                                goto out;
                        }

                }
                ASSERT(parent);
handle_topo_change:

                mutex_enter(&mpt->m_mutex);
                /*
                 * If HBA is being reset, don't perform operations depending
                 * on the IOC. We must free the topo list, however.
                 */
                if (!mpt->m_in_reset)
                        mptsas_handle_topo_change(topo_node, parent);
                else
                        NDBG20(("skipping topo change received during reset"));
                save_node = topo_node;
                topo_node = topo_node->next;
                ASSERT(save_node);
```

```
                kmem_free(save_node, sizeof (mptsas_topo_change_list_t));
                mutex_exit(&mpt->m_mutex);

                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
                    (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) {
                        /*
                         * If direct attached device associated, make sure
                         * reset the parent before start the next one. But
                         * all devices associated with expander shares the
                         * parent.  Also, reset parent if this is for RAID.
                         */
                        parent = NULL;
                }
        }
out:
        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
}

static void
mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
    dev_info_t *parent)
{
        mptsas_target_t *ptgt = NULL;
        mptsas_smp_t    *psmp = NULL;
        mptsas_t        *mpt = (void *)topo_node->mpt;
        uint16_t        devhdl;
        uint16_t        attached_devhdl;
        uint64_t        sas_wwn = 0;
        int             rval = 0;
        uint32_t        page_address;
        uint8_t         phy, flags;
        char            *addr = NULL;
        dev_info_t      *lundip;
        int             circ = 0, circ1 = 0;
        char            attached_wwnstr[MPTSAS_WWN_STRLEN];

        NDBG20(("mptsas%d handle_topo_change enter, devhdl 0x%x,"
            "event 0x%x, flags 0x%x", mpt->m_instance, topo_node->devhdl,
            topo_node->event, topo_node->flags));

        ASSERT(mutex_owned(&mpt->m_mutex));

        switch (topo_node->event) {
        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
        {
                char *phy_mask_name;
                mptsas_phymask_t phymask = 0;

                if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
                        /*
                         * Get latest RAID info.
                         */
                        (void) mptsas_get_raid_info(mpt);
                        ptgt = refhash_linear_search(mpt->m_targets,
                            mptsas_target_eval_devhdl, &topo_node->devhdl);
                        if (ptgt == NULL)
                                break;
                } else {
                        ptgt = (void *)topo_node->object;
                }

                if (ptgt == NULL) {
                        /*
                         * If a Phys Disk was deleted, RAID info needs to be
                         * updated to reflect the new topology.
                         */
                        (void) mptsas_get_raid_info(mpt);

                        /*
                         * Get sas device page 0 by DevHandle to make sure if
                         * SSP/SATA end device exist.
                         */
                        page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
                            MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
                            topo_node->devhdl;

                        rval = mptsas_get_target_device_info(mpt, page_address,
                            &devhdl, &ptgt);
                        if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
                                mptsas_log(mpt, CE_NOTE,
                                    "mptsas_handle_topo_change: target %d is "
                                    "not a SAS/SATA device. \n",
                                    topo_node->devhdl);
                        } else if (rval == DEV_INFO_FAIL_ALLOC) {
                                mptsas_log(mpt, CE_NOTE,
                                    "mptsas_handle_topo_change: could not "
                                    "allocate memory. \n");
                        }
                        /*
                         * If rval is DEV_INFO_PHYS_DISK than there is nothing
                         * else to do, just leave.
                         */
                        if (rval != DEV_INFO_SUCCESS) {
                                return;
                        }
                }

                ASSERT(ptgt->m_devhdl == topo_node->devhdl);

                mutex_exit(&mpt->m_mutex);
                flags = topo_node->flags;

                if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
                        phymask = ptgt->m_addr.mta_phymask;
                        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
                        (void) sprintf(phy_mask_name, "%x", phymask);
                        parent = scsi_hba_iport_find(mpt->m_dip,
                            phy_mask_name);
                        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
                        if (parent == NULL) {
                                mptsas_log(mpt, CE_WARN, "Failed to find a "
                                    "iport for PD, should not happen!");
                                mutex_enter(&mpt->m_mutex);
                                break;
                        }
                }

                if (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
                        ndi_devi_enter(parent, &circ1);
                        (void) mptsas_config_raid(parent, topo_node->devhdl,
                            &lundip);
                        ndi_devi_exit(parent, circ1);
                } else {
                        /*
                         * hold nexus for bus configure
                         */
                        ndi_devi_enter(scsi_vhci_dip, &circ);
                        ndi_devi_enter(parent, &circ1);
                        rval = mptsas_config_target(parent, ptgt);
                        /*
                         * release nexus for bus configure
```

```
                         */
                        ndi_devi_exit(parent, circ1);
                        ndi_devi_exit(scsi_vhci_dip, circ);

                        /*
                         * Add parent's props for SMHBA support
                         */
                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
                                bzero(attached_wwnstr,
                                    sizeof (attached_wwnstr));
                                (void) sprintf(attached_wwnstr, "w%016"PRIx64,
                                    ptgt->m_addr.mta_wwn);
                                if (ddi_prop_update_string(DDI_DEV_T_NONE,
                                    parent,
                                    SCSI_ADDR_PROP_ATTACHED_PORT,
                                    attached_wwnstr)
                                    != DDI_PROP_SUCCESS) {
                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
                                            parent,
                                            SCSI_ADDR_PROP_ATTACHED_PORT);
                                        mptsas_log(mpt, CE_WARN, "Failed to"
                                            "attached-port props");
                                        return;
                                }
                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
                                    MPTSAS_NUM_PHYS, 1) !=
                                    DDI_PROP_SUCCESS) {
                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
                                            parent, MPTSAS_NUM_PHYS);
                                        mptsas_log(mpt, CE_WARN, "Failed to"
                                            " create num-phys props");
                                        return;
                                }

                                /*
                                 * Update PHY info for smhba
                                 */
                                mutex_enter(&mpt->m_mutex);
                                if (mptsas_smhba_phy_init(mpt)) {
                                        mutex_exit(&mpt->m_mutex);
                                        mptsas_log(mpt, CE_WARN, "mptsas phy"
                                            " update failed");
                                        return;
                                }
                                mutex_exit(&mpt->m_mutex);

                                /*
                                 * topo_node->un.physport is really the PHY#
                                 * for direct attached devices
                                 */
                                mptsas_smhba_set_one_phy_props(mpt, parent,
                                    topo_node->un.physport, &attached_devhdl);

                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
                                    MPTSAS_VIRTUAL_PORT, 0) !=
                                    DDI_PROP_SUCCESS) {
                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
                                            parent, MPTSAS_VIRTUAL_PORT);
                                        mptsas_log(mpt, CE_WARN,
                                            "mptsas virtual-port"
                                            "port prop update failed");
                                        return;
                                }
                        }
                }
                mutex_enter(&mpt->m_mutex);
                NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
                    "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
                    ptgt->m_addr.mta_phymask));
                break;
        }
        case MPTSAS_DR_EVENT_OFFLINE_TARGET:
        {
                devhdl = topo_node->devhdl;
                ptgt = refhash_linear_search(mpt->m_targets,
                    mptsas_target_eval_devhdl, &devhdl);
                if (ptgt == NULL)
                        break;

                sas_wwn = ptgt->m_addr.mta_wwn;
                phy = ptgt->m_phynum;

                addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

                if (sas_wwn) {
                        (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
                } else {
                        (void) sprintf(addr, "p%x", phy);
                }
                ASSERT(ptgt->m_devhdl == devhdl);

                if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
                    (topo_node->flags ==
                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
                        /*
                         * Get latest RAID info if RAID volume status changes
                         * or Phys Disk status changes
                         */
                        (void) mptsas_get_raid_info(mpt);
                }
                /*
                 * Abort all outstanding command on the device
                 */
                rval = mptsas_do_scsi_reset(mpt, devhdl);
                if (rval) {
                        NDBG20(("mptsas%d handle_topo_change to reset target "
                            "before offline devhdl:%x, phymask:%x, rval:%x",
                            mpt->m_instance, ptgt->m_devhdl,
                            ptgt->m_addr.mta_phymask, rval));
                }

                mutex_exit(&mpt->m_mutex);

                ndi_devi_enter(scsi_vhci_dip, &circ);
                ndi_devi_enter(parent, &circ1);
                rval = mptsas_offline_target(parent, addr);
                ndi_devi_exit(parent, circ1);
                ndi_devi_exit(scsi_vhci_dip, circ);
                NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
                    "phymask:%x, rval:%x", mpt->m_instance,
                    ptgt->m_devhdl, ptgt->m_addr.mta_phymask, rval));

                kmem_free(addr, SCSI_MAXNAMELEN);

                /*
                 * Clear parent's props for SMHBA support
                 */
                flags = topo_node->flags;
                if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
```

```
                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
                            DDI_PROP_SUCCESS) {
                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
                                    SCSI_ADDR_PROP_ATTACHED_PORT);
                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
                                    "prop update failed");
                                break;
                        }
                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
                            MPTSAS_NUM_PHYS, 0) !=
                            DDI_PROP_SUCCESS) {
                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
                                    MPTSAS_NUM_PHYS);
                                mptsas_log(mpt, CE_WARN, "mptsas num phys "
                                    "prop update failed");
                                break;
                        }
                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
                            MPTSAS_VIRTUAL_PORT, 1) !=
                            DDI_PROP_SUCCESS) {
                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
                                    MPTSAS_VIRTUAL_PORT);
                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
                                    "prop update failed");
                                break;
                        }
                }

                mutex_enter(&mpt->m_mutex);
                ptgt->m_led_status = 0;
                (void) mptsas_flush_led_status(mpt, ptgt);
                if (rval == DDI_SUCCESS) {
                        refhash_remove(mpt->m_targets, ptgt);
                        ptgt = NULL;
                } else {
                        /*
                         * clean DR_INTRANSITION flag to allow I/O down to
                         * PHCI driver since failover finished.
                         * Invalidate the devhdl
                         */
                        ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
                        ptgt->m_tgt_unconfigured = 0;
                        mutex_enter(&mpt->m_tx_waitq_mutex);
                        ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
                        mutex_exit(&mpt->m_tx_waitq_mutex);
                }

                /*
                 * Send SAS IO Unit Control to free the dev handle
                 */
                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
                        rval = mptsas_free_devhdl(mpt, devhdl);

                        NDBG20(("mptsas%d handle_topo_change to remove "
                            "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
                            rval));
                }

                break;
        }
        case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
        {
                devhdl = topo_node->devhdl;
                /*
                 * If this is the remove handle event, do a reset first.
```

```
		 */
		if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
			rval = mptsas_do_scsi_reset(mpt, devhdl);
			if (rval) {
				NDBG20(("mpt%d reset target before remove "
				    "devhdl:%x, rval:%x", mpt->m_instance,
				    devhdl, rval));
			}
		}

		/*
		 * Send SAS IO Unit Control to free the dev handle
		 */
		rval = mptsas_free_devhdl(mpt, devhdl);
		NDBG20(("mptsas%d handle_topo_change to remove "
		    "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
		    rval));
		break;
	}
	case MPTSAS_DR_EVENT_RECONFIG_SMP:
	{
		mptsas_smp_t smp;
		dev_info_t *smpdip;

		devhdl = topo_node->devhdl;

		page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
		    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
		rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
		if (rval != DDI_SUCCESS) {
			mptsas_log(mpt, CE_WARN, "failed to online smp, "
			    "handle %x", devhdl);
			return;
		}

		psmp = mptsas_smp_alloc(mpt, &smp);
		if (psmp == NULL) {
			return;
		}

		mutex_exit(&mpt->m_mutex);
		ndi_devi_enter(parent, &circ1);
		(void) mptsas_online_smp(parent, psmp, &smpdip);
		ndi_devi_exit(parent, circ1);

		mutex_enter(&mpt->m_mutex);
		break;
	}
	case MPTSAS_DR_EVENT_OFFLINE_SMP:
	{
		devhdl = topo_node->devhdl;
		uint32_t dev_info;

		psmp = refhash_linear_search(mpt->m_smp_targets,
		    mptsas_smp_eval_devhdl, &devhdl);
		if (psmp == NULL)
			break;
		/*
		 * The mptsas_smp_t data is released only if the dip is offlined
		 * successfully.
		 */
		mutex_exit(&mpt->m_mutex);

		ndi_devi_enter(parent, &circ1);
		rval = mptsas_offline_smp(parent, psmp, NDI_DEVI_REMOVE);
		ndi_devi_exit(parent, circ1);
```

```

		dev_info = psmp->m_deviceinfo;
		if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
		    DEVINFO_DIRECT_ATTACHED) {
			if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
			    MPTSAS_VIRTUAL_PORT, 1) !=
			    DDI_PROP_SUCCESS) {
				(void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
				    MPTSAS_VIRTUAL_PORT);
				mptsas_log(mpt, CE_WARN, "mptsas virtual port "
				    "prop update failed");
				return;
			}
			/*
			 * Check whether the smp connected to the iport,
			 */
			if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
			    MPTSAS_NUM_PHYS, 0) !=
			    DDI_PROP_SUCCESS) {
				(void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
				    MPTSAS_NUM_PHYS);
				mptsas_log(mpt, CE_WARN, "mptsas num phys"
				    "prop update failed");
				return;
			}
			/*
			 * Clear parent's attached-port props
			 */
			bzero(attached_wwnstr, sizeof (attached_wwnstr));
			if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
			    SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
			    DDI_PROP_SUCCESS) {
				(void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
				    SCSI_ADDR_PROP_ATTACHED_PORT);
				mptsas_log(mpt, CE_WARN, "mptsas attached port "
				    "prop update failed");
				return;
			}
		}

		mutex_enter(&mpt->m_mutex);
		NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
		    "rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
		if (rval == DDI_SUCCESS) {
			refhash_remove(mpt->m_smp_targets, psmp);
		} else {
			psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
		}

		bzero(attached_wwnstr, sizeof (attached_wwnstr));

		break;
	}
	default:
		return;
	}
}

/*
 * Record the event if its type is enabled in mpt instance by ioctl.
 */
static void
mptsas_record_event(void *args)
{
	m_replyh_arg_t			*replyh_arg;
	pMpi2EventNotificationReply_t	eventreply;
	uint32_t			event, rfm;
	mptsas_t			*mpt;
	int				i, j;
	uint16_t			event_data_len;
	boolean_t			sendAEN = FALSE;

	replyh_arg = (m_replyh_arg_t *)args;
	rfm = replyh_arg->rfm;
	mpt = replyh_arg->mpt;

	eventreply = (pMpi2EventNotificationReply_t)
	    (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
	event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);


	/*
	 * Generate a system event to let anyone who cares know that a
	 * LOG_ENTRY_ADDED event has occurred.  This is sent no matter what the
	 * event mask is set to.
	 */
	if (event == MPI2_EVENT_LOG_ENTRY_ADDED) {
		sendAEN = TRUE;
	}

	/*
	 * Record the event only if it is not masked.  Determine which dword
	 * and bit of event mask to test.
	 */
	i = (uint8_t)(event / 32);
	j = (uint8_t)(event % 32);
	if ((i < 4) && ((1 << j) & mpt->m_event_mask[i])) {
		i = mpt->m_event_index;
		mpt->m_events[i].Type = event;
		mpt->m_events[i].Number = ++mpt->m_event_number;
		bzero(mpt->m_events[i].Data, MPTSAS_MAX_EVENT_DATA_LENGTH * 4);
		event_data_len = ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &eventreply->EventDataLength);

		if (event_data_len > 0) {
			/*
			 * Limit data to size in m_event entry
			 */
			if (event_data_len > MPTSAS_MAX_EVENT_DATA_LENGTH) {
				event_data_len = MPTSAS_MAX_EVENT_DATA_LENGTH;
			}
			for (j = 0; j < event_data_len; j++) {
				mpt->m_events[i].Data[j] =
				    ddi_get32(mpt->m_acc_reply_frame_hdl,
				    &(eventreply->EventData[j]));
			}

			/*
			 * check for index wrap-around
			 */
			if (++i == MPTSAS_EVENT_QUEUE_SIZE) {
				i = 0;
			}
			mpt->m_event_index = (uint8_t)i;

			/*
			 * Set flag to send the event.
			 */
			sendAEN = TRUE;
		}
	}
```

```
	/*
	 * Generate a system event if flag is set to let anyone who cares know
	 * that an event has occurred.
	 */
	if (sendAEN) {
		(void) ddi_log_sysevent(mpt->m_dip, DDI_VENDOR_LSI, "MPT_SAS",
		    "SAS", NULL, NULL, DDI_NOSLEEP);
	}
}

#define SMP_RESET_IN_PROGRESS MPI2_EVENT_SAS_TOPO_LR_SMP_RESET_IN_PROGRESS
/*
 * handle sync events from ioc in interrupt
 * return value:
 * DDI_SUCCESS: The event is handled by this func
 * DDI_FAILURE: Event is not handled
 */
static int
mptsas_handle_event_sync(void *args)
{
	m_replyh_arg_t			*replyh_arg;
	pMpi2EventNotificationReply_t	eventreply;
	uint32_t			event, rfm;
	mptsas_t			*mpt;
	uint_t				iocstatus;

	replyh_arg = (m_replyh_arg_t *)args;
	rfm = replyh_arg->rfm;
	mpt = replyh_arg->mpt;

	ASSERT(mutex_owned(&mpt->m_mutex));

	eventreply = (pMpi2EventNotificationReply_t)
	    (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
	event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

	if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
	    &eventreply->IOCStatus)) {
		if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
			mptsas_log(mpt, CE_WARN,
			    "!mptsas_handle_event_sync: event 0x%x, "
			    "IOCStatus=0x%x, "
			    "IOCLogInfo=0x%x", event, iocstatus,
			    ddi_get32(mpt->m_acc_reply_frame_hdl,
			    &eventreply->IOCLogInfo));
		} else {
			mptsas_log(mpt, CE_WARN,
			    "mptsas_handle_event_sync: event 0x%x, "
			    "IOCStatus=0x%x, "
			    "(IOCLogInfo=0x%x)", event, iocstatus,
			    ddi_get32(mpt->m_acc_reply_frame_hdl,
			    &eventreply->IOCLogInfo));
		}
	}

	/*
	 * figure out what kind of event we got and handle accordingly
	 */
	switch (event) {
	case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
	{
		pMpi2EventDataSasTopologyChangeList_t	sas_topo_change_list;
		uint8_t				num_entries, expstatus, phy;
		uint8_t				phystatus, physport, state, i;
		uint8_t				start_phy_num, link_rate;
		uint16_t			dev_handle, reason_code;
		uint16_t			enc_handle, expd_handle;
		char				string[80], curr[80], prev[80];
		mptsas_topo_change_list_t	*topo_head = NULL;
		mptsas_topo_change_list_t	*topo_tail = NULL;
		mptsas_topo_change_list_t	*topo_node = NULL;
		mptsas_target_t			*ptgt;
		mptsas_smp_t			*psmp;
		uint8_t				flags = 0, exp_flag;
		smhba_info_t			*pSmhba = NULL;

		NDBG20(("mptsas_handle_event_sync: SAS topology change"));

		sas_topo_change_list = (pMpi2EventDataSasTopologyChangeList_t)
		    eventreply->EventData;

		enc_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->EnclosureHandle);
		expd_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->ExpanderDevHandle);
		num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->NumEntries);
		start_phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->StartPhyNum);
		expstatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->ExpStatus);
		physport = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_topo_change_list->PhysicalPort);

		string[0] = 0;
		if (expd_handle) {
			flags = MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED;
			switch (expstatus) {
			case MPI2_EVENT_SAS_TOPO_ES_ADDED:
				(void) sprintf(string, " added");
				/*
				 * New expander device added
				 */
				mpt->m_port_chng = 1;
				topo_node = kmem_zalloc(
				    sizeof (mptsas_topo_change_list_t),
				    KM_SLEEP);
				topo_node->mpt = mpt;
				topo_node->event = MPTSAS_DR_EVENT_RECONFIG_SMP;
				topo_node->un.physport = physport;
				topo_node->devhdl = expd_handle;
				topo_node->flags = flags;
				topo_node->object = NULL;
				if (topo_head == NULL) {
					topo_head = topo_tail = topo_node;
				} else {
					topo_tail->next = topo_node;
					topo_tail = topo_node;
				}
				break;
			case MPI2_EVENT_SAS_TOPO_ES_NOT_RESPONDING:
				(void) sprintf(string, " not responding, "
				    "removed");
				psmp = refhash_linear_search(mpt->m_smp_targets,
				    mptsas_smp_eval_devhdl, &expd_handle);
				if (psmp == NULL)
					break;

				topo_node = kmem_zalloc(
				    sizeof (mptsas_topo_change_list_t),
				    KM_SLEEP);
				topo_node->mpt = mpt;
```

```
				topo_node->un.phymask =
				    psmp->m_addr.mta_phymask;
				topo_node->event = MPTSAS_DR_EVENT_OFFLINE_SMP;
				topo_node->devhdl = expd_handle;
				topo_node->flags = flags;
				topo_node->object = NULL;
				if (topo_head == NULL) {
					topo_head = topo_tail = topo_node;
				} else {
					topo_tail->next = topo_node;
					topo_tail = topo_node;
				}
				break;
			case MPI2_EVENT_SAS_TOPO_ES_RESPONDING:
				break;
			case MPI2_EVENT_SAS_TOPO_ES_DELAY_NOT_RESPONDING:
				(void) sprintf(string, " not responding, "
				    "delaying removal");
				break;
			default:
				break;
			}
		} else {
			flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
		}

		NDBG20(("SAS TOPOLOGY CHANGE for enclosure %x expander %x%s\n",
		    enc_handle, expd_handle, string));
		for (i = 0; i < num_entries; i++) {
			phy = i + start_phy_num;
			phystatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
			    &sas_topo_change_list->PHY[i].PhyStatus);
			dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
			    &sas_topo_change_list->PHY[i].AttachedDevHandle);
			reason_code = phystatus & MPI2_EVENT_SAS_TOPO_RC_MASK;
			/*
			 * Filter out processing of Phy Vacant Status unless
			 * the reason code is "Not Responding".  Process all
			 * other combinations of Phy Status and Reason Codes.
			 */
			if ((phystatus &
			    MPI2_EVENT_SAS_TOPO_PHYSTATUS_VACANT) &&
			    (reason_code !=
			    MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING)) {
				continue;
			}
			curr[0] = 0;
			prev[0] = 0;
			string[0] = 0;
			switch (reason_code) {
			case MPI2_EVENT_SAS_TOPO_RC_TARG_ADDED:
			{
				NDBG20(("mptsas%d phy %d physical_port %d "
				    "dev_handle %d added", mpt->m_instance, phy,
				    physport, dev_handle));
				link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
				    &sas_topo_change_list->PHY[i].LinkRate);
				state = (link_rate &
				    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
				    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
				switch (state) {
				case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
					(void) sprintf(curr, "is disabled");
					break;
				case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
					(void) sprintf(curr, "is offline, "
                                    "failed speed negotiation");
                                break;
                        case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
                                (void) sprintf(curr, "SATA OOB "
                                    "complete");
                                break;
                        case SMP_RESET_IN_PROGRESS:
                                (void) sprintf(curr, "SMP reset in "
                                    "progress");
                                break;
                        case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
                                (void) sprintf(curr, "is online at "
                                    "1.5 Gbps");
                                break;
                        case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
                                (void) sprintf(curr, "is online at 3.0 "
                                    "Gbps");
                                break;
                        case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
                                (void) sprintf(curr, "is online at 6.0 "
                                    "Gbps");
                                break;
                        case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
                                (void) sprintf(curr,
                                    "is online at 12.0 Gbps");
                                break;
                        default:
                                (void) sprintf(curr, "state is "
                                    "unknown");
                                break;
                        }
                        /*
                         * New target device added into the system.
                         * Set association flag according to if an
                         * expander is used or not.
                         */
                        exp_flag =
                            MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
                        if (flags ==
                            MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
                                flags = exp_flag;
                        }
                        topo_node = kmem_zalloc(
                            sizeof (mptsas_topo_change_list_t),
                            KM_SLEEP);
                        topo_node->mpt = mpt;
                        topo_node->event =
                            MPTSAS_DR_EVENT_RECONFIG_TARGET;
                        if (expd_handle == 0) {
                                /*
                                 * Per MPI 2, if expander dev handle
                                 * is 0, it's a directly attached
                                 * device. So driver use PHY to decide
                                 * which iport is associated
                                 */
                                physport = phy;
                                mpt->m_port_chng = 1;
                        }
                        topo_node->un.physport = physport;
                        topo_node->devhdl = dev_handle;
                        topo_node->flags = flags;
                        topo_node->object = NULL;
                        if (topo_head == NULL) {
                                topo_head = topo_tail = topo_node;
                        } else {
                                topo_tail->next = topo_node;
```

```
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        case MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING:
                        {
                                NDBG20(("mptsas%d phy %d physical_port %d "
                                    "dev_handle %d removed", mpt->m_instance,
                                    phy, physport, dev_handle));
                                /*
                                 * Set association flag according to if an
                                 * expander is used or not.
                                 */
                                exp_flag =
                                    MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
                                if (flags ==
                                    MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
                                        flags = exp_flag;
                                }
                                /*
                                 * Target device is removed from the system
                                 * Before the device is really offline from
                                 * from system.
                                 */
                                ptgt = refhash_linear_search(mpt->m_targets,
                                    mptsas_target_eval_devhdl, &dev_handle);
                                /*
                                 * If ptgt is NULL here, it means that the
                                 * DevHandle is not in the hash table.  This is
                                 * reasonable sometimes.  For example, if a
                                 * disk was pulled, then added, then pulled
                                 * again, the disk will not have been put into
                                 * the hash table because the add event will
                                 * have an invalid phymask.  BUT, this does not
                                 * mean that the DevHandle is invalid.  The
                                 * controller will still have a valid DevHandle
                                 * that must be removed.  To do this, use the
                                 * MPTSAS_TOPO_FLAG_REMOVE_HANDLE event.
                                 */
                                if (ptgt == NULL) {
                                        topo_node = kmem_zalloc(
                                            sizeof (mptsas_topo_change_list_t),
                                            KM_SLEEP);
                                        topo_node->mpt = mpt;
                                        topo_node->un.phymask = 0;
                                        topo_node->event =
                                            MPTSAS_TOPO_FLAG_REMOVE_HANDLE;
                                        topo_node->devhdl = dev_handle;
                                        topo_node->flags = flags;
                                        topo_node->object = NULL;
                                        if (topo_head == NULL) {
                                                topo_head = topo_tail =
                                                    topo_node;
                                        } else {
                                                topo_tail->next = topo_node;
                                                topo_tail = topo_node;
                                        }
                                        break;
                                }

                                /*
                                 * Update DR flag immediately avoid I/O failure
                                 * before failover finish. Pay attention to the
                                 * mutex protect, we need grab m_tx_waitq_mutex
                                 * during set m_dr_flag because we won't add
                                 * the following command into waitq, instead,
                                 * we need return TRAN_BUSY in the tran_start
                                 * context.
                                 */
                                mutex_enter(&mpt->m_tx_waitq_mutex);
                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
                                mutex_exit(&mpt->m_tx_waitq_mutex);

                                topo_node = kmem_zalloc(
                                    sizeof (mptsas_topo_change_list_t),
                                    KM_SLEEP);
                                topo_node->mpt = mpt;
                                topo_node->un.phymask =
                                    ptgt->m_addr.mta_phymask;
                                topo_node->event =
                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
                                topo_node->devhdl = dev_handle;
                                topo_node->flags = flags;
                                topo_node->object = NULL;
                                if (topo_head == NULL) {
                                        topo_head = topo_tail = topo_node;
                                } else {
                                        topo_tail->next = topo_node;
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        case MPI2_EVENT_SAS_TOPO_RC_PHY_CHANGED:
                                link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
                                    &sas_topo_change_list->PHY[i].LinkRate);
                                state = (link_rate &
                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
                                pSmhba = &mpt->m_phy_info[i].smhba_info;
                                pSmhba->negotiated_link_rate = state;
                                switch (state) {
                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
                                        (void) sprintf(curr, "is disabled");
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_REMOVE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        mpt->m_phy_info[i].smhba_info.
                                            negotiated_link_rate
                                            = 0x1;
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
                                        (void) sprintf(curr, "is offline, "
                                            "failed speed negotiation");
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_OFFLINE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
                                        (void) sprintf(curr, "SATA OOB "
                                            "complete");
                                        break;
                                case SMP_RESET_IN_PROGRESS:
                                        (void) sprintf(curr, "SMP reset in "
                                            "progress");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
                                        (void) sprintf(curr, "is online at "
                                            "1.5 Gbps");
                                        if ((expd_handle == 0) &&
                                            (enc_handle == 1)) {
```

```
                                                mpt->m_port_chng = 1;
                                        }
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_ONLINE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
                                        (void) sprintf(curr, "is online at 3.0 "
                                            "Gbps");
                                        if ((expd_handle == 0) &&
                                            (enc_handle == 1)) {
                                                mpt->m_port_chng = 1;
                                        }
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_ONLINE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
                                        (void) sprintf(curr, "is online at "
                                            "6.0 Gbps");
                                        if ((expd_handle == 0) &&
                                            (enc_handle == 1)) {
                                                mpt->m_port_chng = 1;
                                        }
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_ONLINE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        break;
                                case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
                                        (void) sprintf(curr, "is online at "
                                            "12.0 Gbps");
                                        if ((expd_handle == 0) &&
                                            (enc_handle == 1)) {
                                                mpt->m_port_chng = 1;
                                        }
                                        mptsas_smhba_log_sysevent(mpt,
                                            ESC_SAS_PHY_EVENT,
                                            SAS_PHY_ONLINE,
                                            &mpt->m_phy_info[i].smhba_info);
                                        break;
                                default:
                                        (void) sprintf(curr, "state is "
                                            "unknown");
                                        break;
                                }

                                state = (link_rate &
                                    MPI2_EVENT_SAS_TOPO_LR_PREV_MASK) >>
                                    MPI2_EVENT_SAS_TOPO_LR_PREV_SHIFT;
                                switch (state) {
                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
                                        (void) sprintf(prev, ", was disabled");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
                                        (void) sprintf(prev, ", was offline, "
                                            "failed speed negotiation");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
                                        (void) sprintf(prev, ", was SATA OOB "
                                            "complete");
                                        break;
                                case SMP_RESET_IN_PROGRESS:
                                        (void) sprintf(prev, ", was SMP reset "
                                            "in progress");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
                                        (void) sprintf(prev, ", was online at "
                                            "1.5 Gbps");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
                                        (void) sprintf(prev, ", was online at "
                                            "3.0 Gbps");
                                        break;
                                case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
                                        (void) sprintf(prev, ", was online at "
                                            "6.0 Gbps");
                                        break;
                                case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
                                        (void) sprintf(prev, ", was online at "
                                            "12.0 Gbps");
                                        break;
                                default:
                                break;
                                }
                                (void) sprintf(&string[strlen(string)], "link "
                                    "changed, ");
                                break;
                        case MPI2_EVENT_SAS_TOPO_RC_NO_CHANGE:
                                continue;
                        case MPI2_EVENT_SAS_TOPO_RC_DELAY_NOT_RESPONDING:
                                (void) sprintf(&string[strlen(string)],
                                    "target not responding, delaying "
                                    "removal");
                                break;
                        }
                        NDBG20(("mptsas%d phy %d DevHandle %x, %s%s%s\n",
                            mpt->m_instance, phy, dev_handle, string, curr,
                            prev));
                }
                if (topo_head != NULL) {
                        /*
                         * Launch DR taskq to handle topology change
                         */
                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
                            mptsas_handle_dr, (void *)topo_head,
                            DDI_NOSLEEP)) != DDI_SUCCESS) {
                                while (topo_head != NULL) {
                                        topo_node = topo_head;
                                        topo_head = topo_head->next;
                                        kmem_free(topo_node,
                                            sizeof (mptsas_topo_change_list_t));
                                }
                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
                                    "for handle SAS DR event failed. \n");
                        }
                }
                break;
        }
        case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
        {
                Mpi2EventDataIrConfigChangeList_t       *irChangeList;
                mptsas_topo_change_list_t               *topo_head = NULL;
                mptsas_topo_change_list_t               *topo_tail = NULL;
                mptsas_topo_change_list_t               *topo_node = NULL;
                mptsas_target_t                         *ptgt;
                uint8_t                                 num_entries, i, reason;
                uint16_t                                volhandle, diskhandle;

                irChangeList = (pMpi2EventDataIrConfigChangeList_t)
```

```
                    eventreply->EventData;
                num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
                    &irChangeList->NumElements);

                NDBG20(("mptsas%d IR_CONFIGURATION_CHANGE_LIST event received",
                    mpt->m_instance));

                for (i = 0; i < num_entries; i++) {
                        reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
                            &irChangeList->ConfigElement[i].ReasonCode);
                        volhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
                            &irChangeList->ConfigElement[i].VolDevHandle);
                        diskhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
                            &irChangeList->ConfigElement[i].PhysDiskDevHandle);

                        switch (reason) {
                        case MPI2_EVENT_IR_CHANGE_RC_ADDED:
                        case MPI2_EVENT_IR_CHANGE_RC_VOLUME_CREATED:
                        {
                                NDBG20(("mptsas %d volume added\n",
                                    mpt->m_instance));

                                topo_node = kmem_zalloc(
                                    sizeof (mptsas_topo_change_list_t),
                                    KM_SLEEP);

                                topo_node->mpt = mpt;
                                topo_node->event =
                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
                                topo_node->un.physport = 0xff;
                                topo_node->devhdl = volhandle;
                                topo_node->flags =
                                    MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
                                topo_node->object = NULL;
                                if (topo_head == NULL) {
                                        topo_head = topo_tail = topo_node;
                                } else {
                                        topo_tail->next = topo_node;
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        case MPI2_EVENT_IR_CHANGE_RC_REMOVED:
                        case MPI2_EVENT_IR_CHANGE_RC_VOLUME_DELETED:
                        {
                                NDBG20(("mptsas %d volume deleted\n",
                                    mpt->m_instance));
                                ptgt = refhash_linear_search(mpt->m_targets,
                                    mptsas_target_eval_devhdl, &volhandle);
                                if (ptgt == NULL)
                                        break;

                                /*
                                 * Clear any flags related to volume
                                 */
                                (void) mptsas_delete_volume(mpt, volhandle);

                                /*
                                 * Update DR flag immediately avoid I/O failure
                                 */
                                mutex_enter(&mpt->m_tx_waitq_mutex);
                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
                                mutex_exit(&mpt->m_tx_waitq_mutex);

                                topo_node = kmem_zalloc(
                                    sizeof (mptsas_topo_change_list_t),
                                    KM_SLEEP);
                                topo_node->mpt = mpt;
                                topo_node->un.phymask =
                                    ptgt->m_addr.mta_phymask;
                                topo_node->event =
                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
                                topo_node->devhdl = volhandle;
                                topo_node->flags =
                                    MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
                                topo_node->object = (void *)ptgt;
                                if (topo_head == NULL) {
                                        topo_head = topo_tail = topo_node;
                                } else {
                                        topo_tail->next = topo_node;
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        case MPI2_EVENT_IR_CHANGE_RC_PD_CREATED:
                        case MPI2_EVENT_IR_CHANGE_RC_HIDE:
                        {
                                ptgt = refhash_linear_search(mpt->m_targets,
                                    mptsas_target_eval_devhdl, &diskhandle);
                                if (ptgt == NULL)
                                        break;

                                /*
                                 * Update DR flag immediately avoid I/O failure
                                 */
                                mutex_enter(&mpt->m_tx_waitq_mutex);
                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
                                mutex_exit(&mpt->m_tx_waitq_mutex);

                                topo_node = kmem_zalloc(
                                    sizeof (mptsas_topo_change_list_t),
                                    KM_SLEEP);
                                topo_node->mpt = mpt;
                                topo_node->un.phymask =
                                    ptgt->m_addr.mta_phymask;
                                topo_node->event =
                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
                                topo_node->devhdl = diskhandle;
                                topo_node->flags =
                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
                                topo_node->object = (void *)ptgt;
                                if (topo_head == NULL) {
                                        topo_head = topo_tail = topo_node;
                                } else {
                                        topo_tail->next = topo_node;
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        case MPI2_EVENT_IR_CHANGE_RC_UNHIDE:
                        case MPI2_EVENT_IR_CHANGE_RC_PD_DELETED:
                        {
                                /*
                                 * The physical drive is released by a IR
                                 * volume. But we cannot get the the physport
                                 * or phynum from the event data, so we only
                                 * can get the physport/phynum after SAS
                                 * Device Page0 request for the devhdl.
                                 */
                                topo_node = kmem_zalloc(
                                    sizeof (mptsas_topo_change_list_t),
                                    KM_SLEEP);
```

```
                                topo_node->mpt = mpt;
                                topo_node->un.phymask = 0;
                                topo_node->event =
                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
                                topo_node->devhdl = diskhandle;
                                topo_node->flags =
                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
                                topo_node->object = NULL;
                                mpt->m_port_chng = 1;
                                if (topo_head == NULL) {
                                        topo_head = topo_tail = topo_node;
                                } else {
                                        topo_tail->next = topo_node;
                                        topo_tail = topo_node;
                                }
                                break;
                        }
                        default:
                                break;
                        }
                }

                if (topo_head != NULL) {
                        /*
                         * Launch DR taskq to handle topology change
                         */
                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
                            mptsas_handle_dr, (void *)topo_head,
                            DDI_NOSLEEP)) != DDI_SUCCESS) {
                                while (topo_head != NULL) {
                                        topo_node = topo_head;
                                        topo_head = topo_head->next;
                                        kmem_free(topo_node,
                                            sizeof (mptsas_topo_change_list_t));
                                }
                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
                                    "for handle SAS DR event failed. \n");
                        }
                }
                break;
        }
        default:
                return (DDI_FAILURE);
        }

        return (DDI_SUCCESS);
}

/*
 * handle events from ioc
 */
static void
mptsas_handle_event(void *args)
{
        m_replyh_arg_t                  *replyh_arg;
        pMpi2EventNotificationReply_t   eventreply;
        uint32_t                        event, iocloginfo, rfm;
        uint32_t                        status;
        uint8_t                         port;
        mptsas_t                        *mpt;
        uint_t                          iocstatus;

        replyh_arg = (m_replyh_arg_t *)args;
        rfm = replyh_arg->rfm;
        mpt = replyh_arg->mpt;
	mutex_enter(&mpt->m_mutex);
	/*
	 * If HBA is being reset, drop incoming event.
	 */
	if (mpt->m_in_reset) {
		NDBG20(("dropping event received prior to reset"));
		mutex_exit(&mpt->m_mutex);
		return;
	}

	eventreply = (pMpi2EventNotificationReply_t)
	    (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
	event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

	if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
	    &eventreply->IOCStatus)) {
		if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
			mptsas_log(mpt, CE_WARN,
			    "!mptsas_handle_event: IOCStatus=0x%x, "
			    "IOCLogInfo=0x%x", iocstatus,
			    ddi_get32(mpt->m_acc_reply_frame_hdl,
			    &eventreply->IOCLogInfo));
		} else {
			mptsas_log(mpt, CE_WARN,
			    "mptsas_handle_event: IOCStatus=0x%x, "
			    "IOCLogInfo=0x%x", iocstatus,
			    ddi_get32(mpt->m_acc_reply_frame_hdl,
			    &eventreply->IOCLogInfo));
		}
	}

	/*
	 * figure out what kind of event we got and handle accordingly
	 */
	switch (event) {
	case MPI2_EVENT_LOG_ENTRY_ADDED:
		break;
	case MPI2_EVENT_LOG_DATA:
		iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    &eventreply->IOCLogInfo);
		NDBG20(("mptsas %d log info %x received.\n", mpt->m_instance,
		    iocloginfo));
		break;
	case MPI2_EVENT_STATE_CHANGE:
		NDBG20(("mptsas%d state change.", mpt->m_instance));
		break;
	case MPI2_EVENT_HARD_RESET_RECEIVED:
		NDBG20(("mptsas%d event change.", mpt->m_instance));
		break;
	case MPI2_EVENT_SAS_DISCOVERY:
	{
		MPI2_EVENT_DATA_SAS_DISCOVERY	*sasdiscovery;
		char				string[80];
		uint8_t				rc;

		sasdiscovery =
		    (pMpi2EventDataSasDiscovery_t)eventreply->EventData;

		rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sasdiscovery->ReasonCode);
		port = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sasdiscovery->PhysicalPort);
		status = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    &sasdiscovery->DiscoveryStatus);

		string[0] = 0;
```

```
		switch (rc) {
		case MPI2_EVENT_SAS_DISC_RC_STARTED:
			(void) sprintf(string, "STARTING");
			break;
		case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
			(void) sprintf(string, "COMPLETED");
			break;
		default:
			(void) sprintf(string, "UNKNOWN");
			break;
		}

		NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
		    port, status));

		break;
	}
	case MPI2_EVENT_EVENT_CHANGE:
		NDBG20(("mptsas%d event change.", mpt->m_instance));
		break;
	case MPI2_EVENT_TASK_SET_FULL:
	{
		pMpi2EventDataTaskSetFull_t	taskfull;

		taskfull = (pMpi2EventDataTaskSetFull_t)eventreply->EventData;

		NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
		    mpt->m_instance,  ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &taskfull->CurrentDepth)));
		break;
	}
	case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
	{
		/*
		 * SAS TOPOLOGY CHANGE LIST Event has already been handled
		 * in mptsas_handle_event_sync() of interrupt context
		 */
		break;
	}
	case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
	{
		pMpi2EventDataSasEnclDevStatusChange_t	encstatus;
		uint8_t					rc;
		char					string[80];

		encstatus = (pMpi2EventDataSasEnclDevStatusChange_t)
		    eventreply->EventData;

		rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &encstatus->ReasonCode);
		switch (rc) {
		case MPI2_EVENT_SAS_ENCL_RC_ADDED:
			(void) sprintf(string, "added");
			break;
		case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
			(void) sprintf(string, ", not responding");
			break;
		default:
		break;
		}
		NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure "
		    "%x%s\n", mpt->m_instance,
		    ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &encstatus->EnclosureHandle), string));
		break;
	}
	/*
	 * MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by
	 * mptsas_handle_event_sync,in here just send ack message.
	 */
	case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
	{
		pMpi2EventDataSasDeviceStatusChange_t	statuschange;
		uint8_t					rc;
		uint16_t				devhdl;
		uint64_t				wwn = 0;
		uint32_t				wwn_lo, wwn_hi;

		statuschange = (pMpi2EventDataSasDeviceStatusChange_t)
		    eventreply->EventData;
		rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &statuschange->ReasonCode);
		wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    (uint32_t *)(void *)&statuschange->SASAddress);
		wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    (uint32_t *)(void *)&statuschange->SASAddress + 1);
		wwn = ((uint64_t)wwn_hi << 32) | wwn_lo;
		devhdl =  ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &statuschange->DevHandle);

		NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wwn is %"PRIx64,
		    wwn));

		switch (rc) {
		case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
			NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
			    ddi_get8(mpt->m_acc_reply_frame_hdl,
			    &statuschange->ASC),
			    ddi_get8(mpt->m_acc_reply_frame_hdl,
			    &statuschange->ASCQ)));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
			NDBG20(("Device not supported"));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
			NDBG20(("IOC internally generated the Target Reset "
			    "for devhdl:%x", devhdl));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
			NDBG20(("IOC's internally generated Target Reset "
			    "completed for devhdl:%x", devhdl));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
			NDBG20(("IOC internally generated Abort Task"));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
			NDBG20(("IOC's internally generated Abort Task "
			    "completed"));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
			NDBG20(("IOC internally generated Abort Task Set"));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
			NDBG20(("IOC internally generated Clear Task Set"));
```

```
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
			NDBG20(("IOC internally generated Query Task"));
			break;

		case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
			NDBG20(("Device sent an Asynchronous Notification"));
			break;

		default:
			break;
		}
		break;
	}
	case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
	{
		/*
		 * IR TOPOLOGY CHANGE LIST Event has already been handled
		 * in mpt_handle_event_sync() of interrupt context
		 */
		break;
	}
	case MPI2_EVENT_IR_OPERATION_STATUS:
	{
		Mpi2EventDataIrOperationStatus_t	*irOpStatus;
		char					reason_str[80];
		uint8_t					rc, percent;
		uint16_t				handle;

		irOpStatus = (pMpi2EventDataIrOperationStatus_t)
		    eventreply->EventData;
		rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &irOpStatus->RAIDOperation);
		percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &irOpStatus->PercentComplete);
		handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &irOpStatus->VolDevHandle);

		switch (rc) {
			case MPI2_EVENT_IR_RAIDOP_RESYNC:
				(void) sprintf(reason_str, "resync");
				break;
			case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
				(void) sprintf(reason_str, "online capacity "
				    "expansion");
				break;
			case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
				(void) sprintf(reason_str, "consistency check");
				break;
			default:
				(void) sprintf(reason_str, "unknown reason %x",
				    rc);
		}

		NDBG20(("mptsas%d raid operational status: (%s)"
		    "\thandle(0x%04x), percent complete(%d)\n",
		    mpt->m_instance, reason_str, handle, percent));
		break;
	}
	case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
	{
		pMpi2EventDataSasBroadcastPrimitive_t	sas_broadcast;
		uint8_t					phy_num;
		uint8_t					primitive;
		sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
		    eventreply->EventData;

		phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_broadcast->PhyNum);
		primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
		    &sas_broadcast->Primitive);

		switch (primitive) {
		case MPI2_EVENT_PRIMITIVE_CHANGE:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_CHANGE,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_SES:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_SES,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_EXPANDER:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D01_4,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_ASYNCHRONOUS_EVENT:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D04_7,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_RESERVED3:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D16_7,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_RESERVED4:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D29_7,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_CHANGE0_RESERVED:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D24_0,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		case MPI2_EVENT_PRIMITIVE_CHANGE1_RESERVED:
			mptsas_smhba_log_sysevent(mpt,
			    ESC_SAS_HBA_PORT_BROADCAST,
			    SAS_PORT_BROADCAST_D27_4,
			    &mpt->m_phy_info[phy_num].smhba_info);
			break;
		default:
			NDBG16(("mptsas%d: unknown BROADCAST PRIMITIVE"
			    " %x received",
			    mpt->m_instance, primitive));
			break;
		}
		NDBG16(("mptsas%d sas broadcast primitive: "
		    "\tprimitive(0x%04x), phy(%d) complete\n",
		    mpt->m_instance, primitive, phy_num));
```

```
			break;
	}
	case MPI2_EVENT_IR_VOLUME:
	{
		Mpi2EventDataIrVolume_t		*irVolume;
		uint16_t			devhandle;
		uint32_t			state;
		int				config, vol;
		uint8_t				found = FALSE;

		irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
		state = ddi_get32(mpt->m_acc_reply_frame_hdl,
		    &irVolume->NewValue);
		devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
		    &irVolume->VolDevHandle);

		NDBG20(("EVENT_IR_VOLUME event is received"));

		/*
		 * Get latest RAID info and then find the DevHandle for this
		 * event in the configuration.  If the DevHandle is not found
		 * just exit the event.
		 */
		(void) mptsas_get_raid_info(mpt);
		for (config = 0; (config < mpt->m_num_raid_configs) &&
		    (!found); config++) {
			for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
				if (mpt->m_raidconfig[config].m_raidvol[vol].
				    m_raidhandle == devhandle) {
					found = TRUE;
					break;
				}
			}
		}
		if (!found) {
			break;
		}

		switch (irVolume->ReasonCode) {
		case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
		{
			uint32_t i;
			mpt->m_raidconfig[config].m_raidvol[vol].m_settings =
			    state;

			i = state & MPI2_RAIDVOL0_SETTING_MASK_WRITE_CACHING;
			mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
			    ", auto-config of hot-swap drives is %s"
			    ", write caching is %s"
			    ", hot-spare pool mask is %02x\n",
			    vol, state &
			    MPI2_RAIDVOL0_SETTING_AUTO_CONFIG_HSWAP_DISABLE
			    ? "disabled" : "enabled",
			    i == MPI2_RAIDVOL0_SETTING_UNCHANGED
			    ? "controlled by member disks" :
			    i == MPI2_RAIDVOL0_SETTING_DISABLE_WRITE_CACHING
			    ? "disabled" :
			    i == MPI2_RAIDVOL0_SETTING_ENABLE_WRITE_CACHING
			    ? "enabled" :
			    "incorrectly set",
			    (state >> 16) & 0xff);
				break;
		}
		case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
		{
			mpt->m_raidconfig[config].m_raidvol[vol].m_state =
                                (uint8_t)state;

                            mptsas_log(mpt, CE_NOTE,
                                "Volume %d is now %s\n", vol,
                                state == MPI2_RAID_VOL_STATE_OPTIMAL
                                ? "optimal" :
                                state == MPI2_RAID_VOL_STATE_DEGRADED
                                ? "degraded" :
                                state == MPI2_RAID_VOL_STATE_ONLINE
                                ? "online" :
                                state == MPI2_RAID_VOL_STATE_INITIALIZING
                                ? "initializing" :
                                state == MPI2_RAID_VOL_STATE_FAILED
                                ? "failed" :
                                state == MPI2_RAID_VOL_STATE_MISSING
                                ? "missing" :
                                "state unknown");
                            break;
                    }
                    case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
                    {
                            mpt->m_raidconfig[config].m_raidvol[vol].
                                m_statusflags = state;

                            mptsas_log(mpt, CE_NOTE,
                                " Volume %d is now %s%s%s%s%s%s%s%s%s\n",
                                vol,
                                state & MPI2_RAIDVOL0_STATUS_FLAG_ENABLED
                                ? ", enabled" : ", disabled",
                                state & MPI2_RAIDVOL0_STATUS_FLAG_QUIESCED
                                ? ", quiesced" : "",
                                state & MPI2_RAIDVOL0_STATUS_FLAG_VOLUME_INACTIVE
                                ? ", inactive" : ", active",
                                state &
                                MPI2_RAIDVOL0_STATUS_FLAG_BAD_BLOCK_TABLE_FULL
                                ? ", bad block table is full" : "",
                                state &
                                MPI2_RAIDVOL0_STATUS_FLAG_RESYNC_IN_PROGRESS
                                ? ", resync in progress" : "",
                                state & MPI2_RAIDVOL0_STATUS_FLAG_BACKGROUND_INIT
                                ? ", background initialization in progress" : "",
                                state &
                                MPI2_RAIDVOL0_STATUS_FLAG_CAPACITY_EXPANSION
                                ? ", capacity expansion in progress" : "",
                                state &
                                MPI2_RAIDVOL0_STATUS_FLAG_CONSISTENCY_CHECK
                                ? ", consistency check in progress" : "",
                                state & MPI2_RAIDVOL0_STATUS_FLAG_DATA_SCRUB
                                ? ", data scrub in progress" : "");
                            break;
                    }
                    default:
                            break;
                    }
                    break;
            }
            case MPI2_EVENT_IR_PHYSICAL_DISK:
            {
                    Mpi2EventDataIrPhysicalDisk_t   *irPhysDisk;
                    uint16_t                        devhandle, enchandle, slot;
                    uint32_t                        status, state;
                    uint8_t                         physdisknum, reason;

                    irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
                        eventreply->EventData;
                    physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,
```

```
                        &irPhysDisk->PhysDiskNum);
                    devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
                        &irPhysDisk->PhysDiskDevHandle);
                    enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
                        &irPhysDisk->EnclosureHandle);
                    slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
                        &irPhysDisk->Slot);
                    state = ddi_get32(mpt->m_acc_reply_frame_hdl,
                        &irPhysDisk->NewValue);
                    reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
                        &irPhysDisk->ReasonCode);

                    NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));

                    switch (reason) {
                    case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
                            mptsas_log(mpt, CE_NOTE,
                                " PhysDiskNum %d with DevHandle 0x%x in slot %d "
                                "for enclosure with handle 0x%x is now in hot "
                                "spare pool %d",
                                physdisknum, devhandle, slot, enchandle,
                                (state >> 16) & 0xff);
                            break;

                    case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
                            status = state;
                            mptsas_log(mpt, CE_NOTE,
                                " PhysDiskNum %d with DevHandle 0x%x in slot %d "
                                "for enclosure with handle 0x%x is now "
                                "%s%s%s%s%s\n", physdisknum, devhandle, slot,
                                enchandle,
                                status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
                                ? ", inactive" : ", active",
                                status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
                                ? ", out of sync" : "",
                                status & MPI2_PHYSDISK0_STATUS_FLAG_QUIESCED
                                ? ", quiesced" : "",
                                status &
                                MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
                                ? ", write cache enabled" : "",
                                status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
                                ? ", capacity expansion target" : "");
                            break;

                    case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
                            mptsas_log(mpt, CE_NOTE,
                                " PhysDiskNum %d with DevHandle 0x%x in slot %d "
                                "for enclosure with handle 0x%x is now %s\n",
                                physdisknum, devhandle, slot, enchandle,
                                state == MPI2_RAID_PD_STATE_OPTIMAL
                                ? "optimal" :
                                state == MPI2_RAID_PD_STATE_REBUILDING
                                ? "rebuilding" :
                                state == MPI2_RAID_PD_STATE_DEGRADED
                                ? "degraded" :
                                state == MPI2_RAID_PD_STATE_HOT_SPARE
                                ? "a hot spare" :
                                state == MPI2_RAID_PD_STATE_ONLINE
                                ? "online" :
                                state == MPI2_RAID_PD_STATE_OFFLINE
                                ? "offline" :
                                state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
                                ? "not compatible" :
                                state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
                                ? "not configured" :
                                "state unknown");
                        break;
                }
                break;
        }
        default:
                NDBG20(("mptsas%d: unknown event %x received",
                    mpt->m_instance, event));
                break;
        }

        /*
         * Return the reply frame to the free queue.
         */
        ddi_put32(mpt->m_acc_free_queue_hdl,
            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index], rfm);
        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);
        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                mpt->m_free_index = 0;
        }
        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
            mpt->m_free_index);
        mutex_exit(&mpt->m_mutex);
}

/*
 * invoked from timeout() to restart qfull cmds with throttle == 0
 */
static void
mptsas_restart_cmd(void *arg)
{
        mptsas_t        *mpt = arg;
        mptsas_target_t *ptgt = NULL;

        mutex_enter(&mpt->m_mutex);

        mpt->m_restart_cmd_timeid = 0;

        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                if (ptgt->m_reset_delay == 0) {
                        if (ptgt->m_t_throttle == QFULL_THROTTLE) {
                                mptsas_set_throttle(mpt, ptgt,
                                    MAX_THROTTLE);
                        }
                }
        }
        mptsas_restart_hba(mpt);
        mutex_exit(&mpt->m_mutex);
}

void
mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        int             slot;
        mptsas_slots_t  *slots = mpt->m_active;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

        ASSERT(cmd != NULL);
        ASSERT(cmd->cmd_queued == FALSE);

        /*
         * Task Management cmds are removed in their own routines.  Also,
         * we don't want to modify timeout based on TM cmds.
         */
        if (cmd->cmd_flags & CFLAG_TM_CMD) {
```

```
                return;
        }

        slot = cmd->cmd_slot;

        /*
         * remove the cmd.
         */
        if (cmd == slots->m_slot[slot]) {
                NDBG31(("mptsas_remove_cmd: removing cmd=0x%p, flags "
                    "0x%x", (void *)cmd, cmd->cmd_flags));
                slots->m_slot[slot] = NULL;
                mpt->m_ncmds--;

                /*
                 * only decrement per target ncmds if command
                 * has a target associated with it.
                 */
                if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
                        ptgt->m_t_ncmds--;
                        /*
                         * reset throttle if we just ran an untagged command
                         * to a tagged target
                         */
                        if ((ptgt->m_t_ncmds == 0) &&
                            ((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0)) {
                                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
                        }

                        /*
                         * Remove this command from the active queue.
                         */
                        if (cmd->cmd_active_expiration != 0) {
                                TAILQ_REMOVE(&ptgt->m_active_cmdq, cmd,
                                    cmd_active_link);
                                cmd->cmd_active_expiration = 0;
                        }
                }
        }

        /*
         * This is all we need to do for ioc commands.
         */
        if (cmd->cmd_flags & CFLAG_CMDIOC) {
                mptsas_return_to_pool(mpt, cmd);
                return;
        }

        ASSERT(cmd != slots->m_slot[cmd->cmd_slot]);
}

/*
 * accept all cmds on the tx_waitq if any and then
 * start a fresh request from the top of the device queue.
 *
 * since there are always cmds queued on the tx_waitq, and rare cmds on
 * the instance waitq, so this function should not be invoked in the ISR,
 * the mptsas_restart_waitq() is invoked in the ISR instead. otherwise, the
 * burden belongs to the IO dispatch CPUs is moved the interrupt CPU.
 */
static void
mptsas_restart_hba(mptsas_t *mpt)
{
        ASSERT(mutex_owned(&mpt->m_mutex));

        mutex_enter(&mpt->m_tx_waitq_mutex);
        if (mpt->m_tx_waitq) {
                mptsas_accept_tx_waitq(mpt);
        }
        mutex_exit(&mpt->m_tx_waitq_mutex);
        mptsas_restart_waitq(mpt);
}

/*
 * start a fresh request from the top of the device queue
 */
static void
mptsas_restart_waitq(mptsas_t *mpt)
{
        mptsas_cmd_t    *cmd, *next_cmd;
        mptsas_target_t *ptgt = NULL;

        NDBG1(("mptsas_restart_waitq: mpt=0x%p", (void *)mpt));

        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * If there is a reset delay, don't start any cmds.  Otherwise, start
         * as many cmds as possible.
         * Since SMID 0 is reserved and the TM slot is reserved, the actual max
         * commands is m_max_requests - 2.
         */
        cmd = mpt->m_waitq;

        while (cmd != NULL) {
                next_cmd = cmd->cmd_linkp;
                if (cmd->cmd_flags & CFLAG_PASSTHRU) {
                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                                /*
                                 * passthru command get slot need
                                 * set CFLAG_PREPARED.
                                 */
                                cmd->cmd_flags |= CFLAG_PREPARED;
                                mptsas_waitq_delete(mpt, cmd);
                                mptsas_start_passthru(mpt, cmd);
                        }
                        cmd = next_cmd;
                        continue;
                }
                if (cmd->cmd_flags & CFLAG_CONFIG) {
                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                                /*
                                 * Send the config page request and delete it
                                 * from the waitq.
                                 */
                                cmd->cmd_flags |= CFLAG_PREPARED;
                                mptsas_waitq_delete(mpt, cmd);
                                mptsas_start_config_page_access(mpt, cmd);
                        }
                        cmd = next_cmd;
                        continue;
                }
                if (cmd->cmd_flags & CFLAG_FW_DIAG) {
                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                                /*
                                 * Send the FW Diag request and delete if from
                                 * the waitq.
                                 */
                                cmd->cmd_flags |= CFLAG_PREPARED;
                                mptsas_waitq_delete(mpt, cmd);
                                mptsas_start_diag(mpt, cmd);
                        }
```

```
                        cmd = next_cmd;
                        continue;
                }

                ptgt = cmd->cmd_tgt_addr;
                if (ptgt && (ptgt->m_t_throttle == DRAIN_THROTTLE) &&
                    (ptgt->m_t_ncmds == 0)) {
                        mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
                }
                if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
                    (ptgt && (ptgt->m_reset_delay == 0)) &&
                    (ptgt && (ptgt->m_t_ncmds <
                    ptgt->m_t_throttle))) {
                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                                mptsas_waitq_delete(mpt, cmd);
                                (void) mptsas_start_cmd(mpt, cmd);
                        }
                }
                cmd = next_cmd;
        }
}
/*
 * Cmds are queued if tran_start() doesn't get the m_mutexlock(no wait).
 * Accept all those queued cmds before new cmd is accept so that the
 * cmds are sent in order.
 */
static void
mptsas_accept_tx_waitq(mptsas_t *mpt)
{
        mptsas_cmd_t *cmd;

        ASSERT(mutex_owned(&mpt->m_mutex));
        ASSERT(mutex_owned(&mpt->m_tx_waitq_mutex));

        /*
         * A Bus Reset could occur at any time and flush the tx_waitq,
         * so we cannot count on the tx_waitq to contain even one cmd.
         * And when the m_tx_waitq_mutex is released and run
         * mptsas_accept_pkt(), the tx_waitq may be flushed.
         */
        cmd = mpt->m_tx_waitq;
        for (;;) {
                if ((cmd = mpt->m_tx_waitq) == NULL) {
                        mpt->m_tx_draining = 0;
                        break;
                }
                if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL) {
                        mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
                }
                cmd->cmd_linkp = NULL;
                mutex_exit(&mpt->m_tx_waitq_mutex);
                if (mptsas_accept_pkt(mpt, cmd) != TRAN_ACCEPT)
                        cmn_err(CE_WARN, "mpt: mptsas_accept_tx_waitq: failed "
                            "to accept cmd on queue\n");
                mutex_enter(&mpt->m_tx_waitq_mutex);
        }
}


/*
 * mpt tag type lookup
 */
static char mptsas_tag_lookup[] =
        {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};

static int
```

```c
mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        struct scsi_pkt         *pkt = CMD2PKT(cmd);
        uint32_t                control = 0;
        caddr_t                 mem;
        pMpi2SCSIIORequest_t    io_request;
        ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
        ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;
        mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
        uint16_t                SMID, io_flags = 0;
        uint32_t                request_desc_low, request_desc_high;
        mptsas_cmd_t            *c;

        NDBG1(("mptsas_start_cmd: cmd=0x%p, flags 0x%x", (void *)cmd,
            cmd->cmd_flags));

        /*
         * Set SMID and increment index.  Rollover to 1 instead of 0 if index
         * is at the max.  0 is an invalid SMID, so we call the first index 1.
         */
        SMID = cmd->cmd_slot;

        /*
         * It is possible for back to back device reset to
         * happen before the reset delay has expired.  That's
         * ok, just let the device reset go out on the bus.
         */
        if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
                ASSERT(ptgt->m_reset_delay == 0);
        }

        /*
         * if a non-tagged cmd is submitted to an active tagged target
         * then drain before submitting this cmd; SCSI-2 allows RQSENSE
         * to be untagged
         */
        if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
            (ptgt->m_t_ncmds > 1) &&
            ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
            (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
                if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
                        NDBG23(("target=%d, untagged cmd, start draining\n",
                            ptgt->m_devhdl));

                        if (ptgt->m_reset_delay == 0) {
                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
                        }

                        mptsas_remove_cmd(mpt, cmd);
                        cmd->cmd_pkt_flags |= FLAG_HEAD;
                        mptsas_waitq_add(mpt, cmd);
                }
                return (DDI_FAILURE);
        }

        /*
         * Set correct tag bits.
         */
        if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
                switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
                    FLAG_TAGMASK) >> 12)]) {
                case MSG_SIMPLE_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
                        break;
                case MSG_HEAD_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
```

```c
                        break;
                case MSG_ORDERED_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
                        break;
                default:
                        mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
                        break;
                }
        } else {
                if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
                                ptgt->m_t_throttle = 1;
                }
                control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
        }

        if (cmd->cmd_pkt_flags & FLAG_TLR) {
                control |= MPI2_SCSIIO_CONTROL_TLR_ON;
        }

        mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
        io_request = (pMpi2SCSIIORequest_t)mem;

        bzero(io_request, sizeof (Mpi2SCSIIORequest_t));
        ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
            (MPI2_SCSI_IO_REQUEST, SGL) / 4);
        mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
            MPI2_FUNCTION_SCSI_IO_REQUEST);

        (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
            io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);

        io_flags = cmd->cmd_cdblen;
        if (mptsas_use_fastpath &&
            ptgt->m_io_flags & MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
                io_flags |= MPI25_SCSIIO_IOFLAGS_FAST_PATH;
                request_desc_low = MPI25_REQ_DESCRIPT_FLAGS_FAST_PATH_SCSI_IO;
        } else {
                request_desc_low = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
        }
        ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
        /*
         * setup the Scatter/Gather DMA list for this request
         */
        if (cmd->cmd_cookiec > 0) {
                mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
        } else {
                ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
                    ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
                    MPI2_SGE_FLAGS_END_OF_BUFFER |
                    MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
                    MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
        }

        /*
         * save ARQ information
         */
        ddi_put8(acc_hdl, &io_request->SenseBufferLength, cmd->cmd_rqslen);
        if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
            (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
                ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
                    cmd->cmd_ext_arqcookie.dmac_address);
        } else {
                ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
                    cmd->cmd_arqcookie.dmac_address);
        }
```

```
	ddi_put32(acc_hdl, &io_request->Control, control);

	NDBG31(("starting message=%d(0x%p), with cmd=0x%p",
	    SMID, (void *)io_request, (void *)cmd));

	(void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);

	/*
	 * Build request descriptor and write it to the request desc post reg.
	 */
	request_desc_low |= (SMID << 16);
	request_desc_high = ptgt->m_devhdl << 16;
	MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);

	/*
	 * Start timeout.
	 */
	cmd->cmd_active_expiration =
	    gethrtime() + (hrtime_t)pkt->pkt_time * NANOSEC;
#ifdef MPTSAS_TEST
	/*
	 * Force timeouts to happen immediately.
	 */
	if (mptsas_test_timeouts)
		cmd->cmd_active_expiration = gethrtime();
#endif
	c = TAILQ_FIRST(&ptgt->m_active_cmdq);
	if (c == NULL ||
	    c->cmd_active_expiration < cmd->cmd_active_expiration) {
		/*
		 * Common case is that this is the last pending expiration
		 * (or queue is empty). Insert at head of the queue.
		 */
		TAILQ_INSERT_HEAD(&ptgt->m_active_cmdq, cmd, cmd_active_link);
	} else {
		/*
		 * Queue is not empty and first element expires later than
		 * this command. Search for element expiring sooner.
		 */
		while ((c = TAILQ_NEXT(c, cmd_active_link)) != NULL) {
			if (c->cmd_active_expiration <
			    cmd->cmd_active_expiration) {
				TAILQ_INSERT_BEFORE(c, cmd, cmd_active_link);
				break;
			}
		}
		if (c == NULL) {
			/*
			 * No element found expiring sooner, append to
			 * non-empty queue.
			 */
			TAILQ_INSERT_TAIL(&ptgt->m_active_cmdq, cmd,
			    cmd_active_link);
		}
	}

	if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
		return (DDI_FAILURE);
	}
	return (DDI_SUCCESS);
}

/*
 * Select a helper thread to handle current doneq
```

```
 */
static void
mptsas_deliver_doneq_thread(mptsas_t *mpt)
{
	uint64_t			t, i;
	uint32_t			min = 0xffffffff;
	mptsas_doneq_thread_list_t	*item;

	for (i = 0; i < mpt->m_doneq_thread_n; i++) {
		item = &mpt->m_doneq_thread_id[i];
		/*
		 * If the completed command on help thread[i] less than
		 * doneq_thread_threshold, then pick the thread[i]. Otherwise
		 * pick a thread which has least completed command.
		 */

		mutex_enter(&item->mutex);
		if (item->len < mpt->m_doneq_thread_threshold) {
			t = i;
			mutex_exit(&item->mutex);
			break;
		}
		if (item->len < min) {
			min = item->len;
			t = i;
		}
		mutex_exit(&item->mutex);
	}
	mutex_enter(&mpt->m_doneq_thread_id[t].mutex);
	mptsas_doneq_mv(mpt, t);
	cv_signal(&mpt->m_doneq_thread_id[t].cv);
	mutex_exit(&mpt->m_doneq_thread_id[t].mutex);
}

/*
 * move the current global doneq to the doneq of thead[t]
 */
static void
mptsas_doneq_mv(mptsas_t *mpt, uint64_t t)
{
	mptsas_cmd_t			*cmd;
	mptsas_doneq_thread_list_t	*item = &mpt->m_doneq_thread_id[t];

	ASSERT(mutex_owned(&item->mutex));
	while ((cmd = mpt->m_doneq) != NULL) {
		if ((mpt->m_doneq = cmd->cmd_linkp) == NULL) {
			mpt->m_donetail = &mpt->m_doneq;
		}
		cmd->cmd_linkp = NULL;
		*item->donetail = cmd;
		item->donetail = &cmd->cmd_linkp;
		mpt->m_doneq_len--;
		item->len++;
	}
}

void
mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	struct scsi_pkt *pkt = CMD2PKT(cmd);

	/* Check all acc and dma handles */
	if ((mptsas_check_acc_handle(mpt->m_datap) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(mpt->m_config_handle) !=
	    DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip,
		    DDI_SERVICE_UNAFFECTED);
		ddi_fm_acc_err_clear(mpt->m_config_handle,
		    DDI_FME_VER0);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
	if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=
	    DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip,
		    DDI_SERVICE_UNAFFECTED);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
	if (cmd->cmd_dmahandle &&
	    (mptsas_check_dma_handle(cmd->cmd_dmahandle) != DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
	if ((cmd->cmd_extra_frames &&
	    ((mptsas_check_dma_handle(cmd->cmd_extra_frames->m_dma_hdl) !=
	    DDI_SUCCESS) ||
	    (mptsas_check_acc_handle(cmd->cmd_extra_frames->m_acc_hdl) !=
	    DDI_SUCCESS)))) {
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
	if (cmd->cmd_arqhandle &&
	    (mptsas_check_dma_handle(cmd->cmd_arqhandle) != DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
	if (cmd->cmd_ext_arqhandle &&
	    (mptsas_check_dma_handle(cmd->cmd_ext_arqhandle) != DDI_SUCCESS)) {
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
		pkt->pkt_reason = CMD_TRAN_ERR;
		pkt->pkt_statistics = 0;
	}
}

/*
 * These routines manipulate the queue of commands that
 * are waiting for their completion routines to be called.
 * The queue is usually in FIFO order but on an MP system
 * it’s possible for the completion routines to get out
```

```
 * of order. If that’s a problem you need to add a global
 * mutex around the code that calls the completion routine
 * in the interrupt handler.
 */
static void
mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	struct scsi_pkt *pkt = CMD2PKT(cmd);

	NDBG31(("mptsas_doneq_add: cmd=0x%p", (void *)cmd));

	ASSERT((cmd->cmd_flags & CFLAG_COMPLETED) == 0);
	cmd->cmd_linkp = NULL;
	cmd->cmd_flags |= CFLAG_FINISHED;
	cmd->cmd_flags &= ~CFLAG_IN_TRANSPORT;

	mptsas_fma_check(mpt, cmd);

	/*
	 * only add scsi pkts that have completion routines to
	 * the doneq.  no intr cmds do not have callbacks.
	 */
	if (pkt && (pkt->pkt_comp)) {
		*mpt->m_donetail = cmd;
		mpt->m_donetail = &cmd->cmd_linkp;
		mpt->m_doneq_len++;
	}
}

static mptsas_cmd_t *
mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t)
{
	mptsas_cmd_t			*cmd;
	mptsas_doneq_thread_list_t	*item = &mpt->m_doneq_thread_id[t];

	/* pop one off the done queue */
	if ((cmd = item->doneq) != NULL) {
		/* if the queue is now empty fix the tail pointer */
		NDBG31(("mptsas_doneq_thread_rm: cmd=0x%p", (void *)cmd));
		if ((item->doneq = cmd->cmd_linkp) == NULL) {
			item->donetail = &item->doneq;
		}
		cmd->cmd_linkp = NULL;
		item->len--;
	}
	return (cmd);
}

static void
mptsas_doneq_empty(mptsas_t *mpt)
{
	if (mpt->m_doneq && !mpt->m_in_callback) {
		mptsas_cmd_t	*cmd, *next;
		struct scsi_pkt *pkt;

		mpt->m_in_callback = 1;
		cmd = mpt->m_doneq;
		mpt->m_doneq = NULL;
		mpt->m_donetail = &mpt->m_doneq;
		mpt->m_doneq_len = 0;

		mutex_exit(&mpt->m_mutex);
		/*
		 * run the completion routines of all the
		 * completed commands
		 */
		while (cmd != NULL) {
			next = cmd->cmd_linkp;
			cmd->cmd_linkp = NULL;
			/* run this command's completion routine */
			cmd->cmd_flags |= CFLAG_COMPLETED;
			pkt = CMD2PKT(cmd);
			mptsas_pkt_comp(pkt, cmd);
			cmd = next;
		}
		mutex_enter(&mpt->m_mutex);
		mpt->m_in_callback = 0;
	}
}

/*
 * These routines manipulate the target's queue of pending requests
 */
void
mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	NDBG7(("mptsas_waitq_add: cmd=0x%p", (void *)cmd));
	mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
	cmd->cmd_queued = TRUE;
	if (ptgt)
		ptgt->m_t_nwait++;
	if (cmd->cmd_pkt_flags & FLAG_HEAD) {
		if ((cmd->cmd_linkp = mpt->m_waitq) == NULL) {
			mpt->m_waitqtail = &cmd->cmd_linkp;
		}
		mpt->m_waitq = cmd;
	} else {
		cmd->cmd_linkp = NULL;
		*(mpt->m_waitqtail) = cmd;
		mpt->m_waitqtail = &cmd->cmd_linkp;
	}
}

static mptsas_cmd_t *
mptsas_waitq_rm(mptsas_t *mpt)
{
	mptsas_cmd_t	*cmd;
	mptsas_target_t *ptgt;
	NDBG7(("mptsas_waitq_rm"));

	MPTSAS_WAITQ_RM(mpt, cmd);

	NDBG7(("mptsas_waitq_rm: cmd=0x%p", (void *)cmd));
	if (cmd) {
		ptgt = cmd->cmd_tgt_addr;
		if (ptgt) {
			ptgt->m_t_nwait--;
			ASSERT(ptgt->m_t_nwait >= 0);
		}
	}
	return (cmd);
}

/*
 * remove specified cmd from the middle of the wait queue.
 */
static void
mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	mptsas_cmd_t	*prevp = mpt->m_waitq;
	mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
```

```
	NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
	    (void *)mpt, (void *)cmd));
	if (ptgt) {
		ptgt->m_t_nwait--;
		ASSERT(ptgt->m_t_nwait >= 0);
	}

	if (prevp == cmd) {
		if ((mpt->m_waitq = cmd->cmd_linkp) == NULL)
			mpt->m_waitqtail = &mpt->m_waitq;

		cmd->cmd_linkp = NULL;
		cmd->cmd_queued = FALSE;
		NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
		    (void *)mpt, (void *)cmd));
		return;
	}

	while (prevp != NULL) {
		if (prevp->cmd_linkp == cmd) {
			if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
				mpt->m_waitqtail = &prevp->cmd_linkp;

			cmd->cmd_linkp = NULL;
			cmd->cmd_queued = FALSE;
			NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
			    (void *)mpt, (void *)cmd));
			return;
		}
		prevp = prevp->cmd_linkp;
	}
	cmn_err(CE_PANIC, "mpt: mptsas_waitq_delete: queue botch");
}

static mptsas_cmd_t *
mptsas_tx_waitq_rm(mptsas_t *mpt)
{
	mptsas_cmd_t *cmd;
	NDBG7(("mptsas_tx_waitq_rm"));

	MPTSAS_TX_WAITQ_RM(mpt, cmd);

	NDBG7(("mptsas_tx_waitq_rm: cmd=0x%p", (void *)cmd));

	return (cmd);
}

/*
 * remove specified cmd from the middle of the tx_waitq.
 */
static void
mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
	mptsas_cmd_t *prevp = mpt->m_tx_waitq;

	NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
	    (void *)mpt, (void *)cmd));

	if (prevp == cmd) {
		if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL)
			mpt->m_tx_waitqtail = &mpt->m_tx_waitq;

		cmd->cmd_linkp = NULL;
		cmd->cmd_queued = FALSE;
		NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
		    (void *)mpt, (void *)cmd));
```

```
                return;
        }

        while (prevp != NULL) {
                if (prevp->cmd_linkp == cmd) {
                        if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
                                mpt->m_tx_waitqtail = &prevp->cmd_linkp;

                        cmd->cmd_linkp = NULL;
                        cmd->cmd_queued = FALSE;
                        NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
                            (void *)mpt, (void *)cmd));
                        return;
                }
                prevp = prevp->cmd_linkp;
        }
        cmn_err(CE_PANIC, "mpt: mptsas_tx_waitq_delete: queue botch");
}

/*
 * device and bus reset handling
 *
 * Notes:
 *      - RESET_ALL:    reset the controller
 *      - RESET_TARGET: reset the target specified in scsi_address
 */
static int
mptsas_scsi_reset(struct scsi_address *ap, int level)
{
        mptsas_t                *mpt = ADDR2MPT(ap);
        int                     rval;
        mptsas_tgt_private_t    *tgt_private;
        mptsas_target_t         *ptgt = NULL;

        tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->tran_tgt_private;
        ptgt = tgt_private->t_private;
        if (ptgt == NULL) {
                return (FALSE);
        }
        NDBG22(("mptsas_scsi_reset: target=%d level=%d", ptgt->m_devhdl,
            level));

        mutex_enter(&mpt->m_mutex);
        /*
         * if we are not in panic set up a reset delay for this target
         */
        if (!ddi_in_panic()) {
                mptsas_setup_bus_reset_delay(mpt);
        } else {
                drv_usecwait(mpt->m_scsi_reset_delay * 1000);
        }
        rval = mptsas_do_scsi_reset(mpt, ptgt->m_devhdl);
        mutex_exit(&mpt->m_mutex);

        /*
         * The transport layer expect to only see TRUE and
         * FALSE. Therefore, we will adjust the return value
         * if mptsas_do_scsi_reset returns FAILED.
         */
        if (rval == FAILED)
                rval = FALSE;
        return (rval);
}

static int
mptsas_do_scsi_reset(mptsas_t *mpt, uint16_t devhdl)
```

```
{
        int             rval = FALSE;
        uint8_t         config, disk;

        ASSERT(mutex_owned(&mpt->m_mutex));

        if (mptsas_debug_resets) {
                mptsas_log(mpt, CE_WARN, "mptsas_do_scsi_reset: target=%d",
                    devhdl);
        }

        /*
         * Issue a Target Reset message to the target specified but not to a
         * disk making up a raid volume.  Just look through the RAID config
         * Phys Disk list of DevHandles.  If the target's DevHandle is in this
         * list, then don't reset this target.
         */
        for (config = 0; config < mpt->m_num_raid_configs; config++) {
                for (disk = 0; disk < MPTSAS_MAX_DISKS_IN_CONFIG; disk++) {
                        if (devhdl == mpt->m_raidconfig[config].
                            m_physdisk_devhdl[disk]) {
                                return (TRUE);
                        }
                }
        }

        rval = mptsas_ioc_task_management(mpt,
            MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET, devhdl, 0, NULL, 0, 0);

        mptsas_doneq_empty(mpt);
        return (rval);
}

static int
mptsas_scsi_reset_notify(struct scsi_address *ap, int flag,
        void (*callback)(caddr_t), caddr_t arg)
{
        mptsas_t        *mpt = ADDR2MPT(ap);

        NDBG22(("mptsas_scsi_reset_notify: tgt=%d", ap->a_target));

        return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
            &mpt->m_mutex, &mpt->m_reset_notify_listf));
}

static int
mptsas_get_name(struct scsi_device *sd, char *name, int len)
{
        dev_info_t      *lun_dip = NULL;

        ASSERT(sd != NULL);
        ASSERT(name != NULL);
        lun_dip = sd->sd_dev;
        ASSERT(lun_dip != NULL);

        if (mptsas_name_child(lun_dip, name, len) == DDI_SUCCESS) {
                return (1);
        } else {
                return (0);
        }
}

static int
mptsas_get_bus_addr(struct scsi_device *sd, char *name, int len)
{
        return (mptsas_get_name(sd, name, len));
```

```
}

void
mptsas_set_throttle(mptsas_t *mpt, mptsas_target_t *ptgt, int what)
{

        NDBG25(("mptsas_set_throttle: throttle=%x", what));

        /*
         * if the bus is draining/quiesced, no changes to the throttles
         * are allowed. Not allowing change of throttles during draining
         * limits error recovery but will reduce draining time
         *
         * all throttles should have been set to HOLD_THROTTLE
         */
        if (mpt->m_softstate & (MPTSAS_SS_QUIESCED | MPTSAS_SS_DRAINING)) {
                return;
        }

        if (what == HOLD_THROTTLE) {
                ptgt->m_t_throttle = HOLD_THROTTLE;
        } else if (ptgt->m_reset_delay == 0) {
                ptgt->m_t_throttle = what;
        }
}

/*
 * Clean up from a device reset.
 * For the case of target reset, this function clears the waitq of all
 * commands for a particular target.   For the case of abort task set, this
 * function clears the waitq of all commonds for a particular target/lun.
 */
static void
mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
{
        mptsas_slots_t  *slots = mpt->m_active;
        mptsas_cmd_t    *cmd, *next_cmd;
        int             slot;
        uchar_t         reason;
        uint_t          stat;
        hrtime_t        timestamp;

        NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));

        timestamp = gethrtime();

        /*
         * Make sure the I/O Controller has flushed all cmds
         * that are associated with this target for a target reset
         * and target/lun for abort task set.
         * Account for TM requests, which use the last SMID.
         */
        for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
                if ((cmd = slots->m_slot[slot]) == NULL)
                        continue;
                reason = CMD_RESET;
                stat = STAT_DEV_RESET;
                switch (tasktype) {
                case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
                        if (Tgt(cmd) == target) {
                                if (cmd->cmd_active_expiration <= timestamp) {
                                        /*
                                         * When timeout requested, propagate
                                         * proper reason and statistics to
                                         * target drivers.
                                         */
```

```
                                        reason = CMD_TIMEOUT;
                                        stat |= STAT_TIMEOUT;
                                }
                                NDBG25(("mptsas_flush_target discovered non-"
                                    "NULL cmd in slot %d, tasktype 0x%x", slot,
                                    tasktype));
                                mptsas_dump_cmd(mpt, cmd);
                                mptsas_remove_cmd(mpt, cmd);
                                mptsas_set_pkt_reason(mpt, cmd, reason, stat);
                                mptsas_doneq_add(mpt, cmd);
                        }
                        break;
                case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
                        reason = CMD_ABORTED;
                        stat = STAT_ABORTED;
                        /*FALLTHROUGH*/
                case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
                        if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {

                                NDBG25(("mptsas_flush_target discovered non-"
                                    "NULL cmd in slot %d, tasktype 0x%x", slot,
                                    tasktype));
                                mptsas_dump_cmd(mpt, cmd);
                                mptsas_remove_cmd(mpt, cmd);
                                mptsas_set_pkt_reason(mpt, cmd, reason,
                                    stat);
                                mptsas_doneq_add(mpt, cmd);
                        }
                        break;
                default:
                        break;
                }
        }

        /*
         * Flush the waitq and tx_waitq of this target's cmds
         */
        cmd = mpt->m_waitq;

        reason = CMD_RESET;
        stat = STAT_DEV_RESET;

        switch (tasktype) {
        case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
                while (cmd != NULL) {
                        next_cmd = cmd->cmd_linkp;
                        if (Tgt(cmd) == target) {
                                mptsas_waitq_delete(mpt, cmd);
                                mptsas_set_pkt_reason(mpt, cmd,
                                    reason, stat);
                                mptsas_doneq_add(mpt, cmd);
                        }
                        cmd = next_cmd;
                }
                mutex_enter(&mpt->m_tx_waitq_mutex);
                cmd = mpt->m_tx_waitq;
                while (cmd != NULL) {
                        next_cmd = cmd->cmd_linkp;
                        if (Tgt(cmd) == target) {
                                mptsas_tx_waitq_delete(mpt, cmd);
                                mutex_exit(&mpt->m_tx_waitq_mutex);
                                mptsas_set_pkt_reason(mpt, cmd,
                                    reason, stat);
                                mptsas_doneq_add(mpt, cmd);
                                mutex_enter(&mpt->m_tx_waitq_mutex);
                        }
                        cmd = next_cmd;
                }
                mutex_exit(&mpt->m_tx_waitq_mutex);
                break;
        case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
                reason = CMD_ABORTED;
                stat =  STAT_ABORTED;
                /*FALLTHROUGH*/
        case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
                while (cmd != NULL) {
                        next_cmd = cmd->cmd_linkp;
                        if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
                                mptsas_waitq_delete(mpt, cmd);
                                mptsas_set_pkt_reason(mpt, cmd,
                                    reason, stat);
                                mptsas_doneq_add(mpt, cmd);
                        }
                        cmd = next_cmd;
                }
                mutex_enter(&mpt->m_tx_waitq_mutex);
                cmd = mpt->m_tx_waitq;
                while (cmd != NULL) {
                        next_cmd = cmd->cmd_linkp;
                        if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
                                mptsas_tx_waitq_delete(mpt, cmd);
                                mutex_exit(&mpt->m_tx_waitq_mutex);
                                mptsas_set_pkt_reason(mpt, cmd,
                                    reason, stat);
                                mptsas_doneq_add(mpt, cmd);
                                mutex_enter(&mpt->m_tx_waitq_mutex);
                        }
                        cmd = next_cmd;
                }
                mutex_exit(&mpt->m_tx_waitq_mutex);
                break;
        default:
                mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
                    tasktype);
                break;
        }
}

/*
 * Clean up hba state, abort all outstanding command and commands in waitq
 * reset timeout of all targets.
 */
static void
mptsas_flush_hba(mptsas_t *mpt)
{
        mptsas_slots_t  *slots = mpt->m_active;
        mptsas_cmd_t    *cmd;
        int             slot;

        NDBG25(("mptsas_flush_hba"));

        /*
         * The I/O Controller should have already sent back
         * all commands via the scsi I/O reply frame.  Make
         * sure all commands have been flushed.
         * Account for TM request, which use the last SMID.
         */
        for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
                if ((cmd = slots->m_slot[slot]) == NULL)
                        continue;

                if (cmd->cmd_flags & CFLAG_CMDIOC) {
```

```
                        /*
                         * Need to make sure to tell everyone that might be
                         * waiting on this command that it's going to fail.  If
                         * we get here, this command will never timeout because
                         * the active command table is going to be re-allocated,
                         * so there will be nothing to check against a time out.
                         * Instead, mark the command as failed due to reset.
                         */
                        mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
                            STAT_BUS_RESET);
                        if ((cmd->cmd_flags &
                            (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG))) {
                                cmd->cmd_flags |= CFLAG_FINISHED;
                                cv_broadcast(&mpt->m_passthru_cv);
                                cv_broadcast(&mpt->m_config_cv);
                                cv_broadcast(&mpt->m_fw_diag_cv);
                        }
                        continue;
                }

                NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
                    slot));
                mptsas_dump_cmd(mpt, cmd);

                mptsas_remove_cmd(mpt, cmd);
                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
                mptsas_doneq_add(mpt, cmd);
        }

        /*
         * Flush the waitq.
         */
        while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
                if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
                    (cmd->cmd_flags & CFLAG_CONFIG) ||
                    (cmd->cmd_flags & CFLAG_FW_DIAG)) {
                        cmd->cmd_flags |= CFLAG_FINISHED;
                        cv_broadcast(&mpt->m_passthru_cv);
                        cv_broadcast(&mpt->m_config_cv);
                        cv_broadcast(&mpt->m_fw_diag_cv);
                } else {
                        mptsas_doneq_add(mpt, cmd);
                }
        }

        /*
         * Flush the tx_waitq
         */
        mutex_enter(&mpt->m_tx_waitq_mutex);
        while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
                mutex_exit(&mpt->m_tx_waitq_mutex);
                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
                mptsas_doneq_add(mpt, cmd);
                mutex_enter(&mpt->m_tx_waitq_mutex);
        }
        mutex_exit(&mpt->m_tx_waitq_mutex);

        /*
         * Drain the taskqs prior to reallocating resources.
         */
        mutex_exit(&mpt->m_mutex);
        ddi_taskq_wait(mpt->m_event_taskq);
        ddi_taskq_wait(mpt->m_dr_taskq);
        mutex_enter(&mpt->m_mutex);
}
```

```
/*
 * set pkt_reason and OR in pkt_statistics flag
 */
static void
mptsas_set_pkt_reason(mptsas_t *mpt, mptsas_cmd_t *cmd, uchar_t reason,
    uint_t stat)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif

        NDBG25(("mptsas_set_pkt_reason: cmd=0x%p reason=%x stat=%x",
            (void *)cmd, reason, stat));

        if (cmd) {
                if (cmd->cmd_pkt->pkt_reason == CMD_CMPLT) {
                        cmd->cmd_pkt->pkt_reason = reason;
                }
                cmd->cmd_pkt->pkt_statistics |= stat;
        }
}

static void
mptsas_start_watch_reset_delay()
{
        NDBG22(("mptsas_start_watch_reset_delay"));

        mutex_enter(&mptsas_global_mutex);
        if (mptsas_reset_watch == NULL && mptsas_timeouts_enabled) {
                mptsas_reset_watch = timeout(mptsas_watch_reset_delay, NULL,
                    drv_usectohz((clock_t)
                    MPTSAS_WATCH_RESET_DELAY_TICK * 1000));
                ASSERT(mptsas_reset_watch != NULL);
        }
        mutex_exit(&mptsas_global_mutex);
}

static void
mptsas_setup_bus_reset_delay(mptsas_t *mpt)
{
        mptsas_target_t *ptgt = NULL;

        ASSERT(MUTEX_HELD(&mpt->m_mutex));

        NDBG22(("mptsas_setup_bus_reset_delay"));
        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
                ptgt->m_reset_delay = mpt->m_scsi_reset_delay;
        }

        mptsas_start_watch_reset_delay();
}

/*
 * mptsas_watch_reset_delay(_subr) is invoked by timeout() and checks every
 * mpt instance for active reset delays
 */
static void
mptsas_watch_reset_delay(void *arg)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(arg))
#endif
```

```
        mptsas_t        *mpt;
        int             not_done = 0;

        NDBG22(("mptsas_watch_reset_delay"));

        mutex_enter(&mptsas_global_mutex);
        mptsas_reset_watch = 0;
        mutex_exit(&mptsas_global_mutex);
        rw_enter(&mptsas_global_rwlock, RW_READER);
        for (mpt = mptsas_head; mpt != NULL; mpt = mpt->m_next) {
                if (mpt->m_tran == 0) {
                        continue;
                }
                mutex_enter(&mpt->m_mutex);
                not_done += mptsas_watch_reset_delay_subr(mpt);
                mutex_exit(&mpt->m_mutex);
        }
        rw_exit(&mptsas_global_rwlock);

        if (not_done) {
                mptsas_start_watch_reset_delay();
        }
}

static int
mptsas_watch_reset_delay_subr(mptsas_t *mpt)
{
        int             done = 0;
        int             restart = 0;
        mptsas_target_t *ptgt = NULL;

        NDBG22(("mptsas_watch_reset_delay_subr: mpt=0x%p", (void *)mpt));

        ASSERT(mutex_owned(&mpt->m_mutex));

        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                if (ptgt->m_reset_delay != 0) {
                        ptgt->m_reset_delay -=
                            MPTSAS_WATCH_RESET_DELAY_TICK;
                        if (ptgt->m_reset_delay <= 0) {
                                ptgt->m_reset_delay = 0;
                                mptsas_set_throttle(mpt, ptgt,
                                    MAX_THROTTLE);
                                restart++;
                        } else {
                                done = -1;
                        }
                }
        }

        if (restart > 0) {
                mptsas_restart_hba(mpt);
        }
        return (done);
}

#ifdef MPTSAS_TEST
static void
mptsas_test_reset(mptsas_t *mpt, int target)
{
        mptsas_target_t    *ptgt = NULL;

        if (mptsas_rtest == target) {
                if (mptsas_do_scsi_reset(mpt, target) == TRUE) {
                        mptsas_rtest = -1;
                }
                if (mptsas_rtest == -1) {
                        NDBG22(("mptsas_test_reset success"));
                }
        }
}
#endif

/*
 * abort handling:
 *
 * Notes:
 *      - if pkt is not NULL, abort just that command
 *      - if pkt is NULL, abort all outstanding commands for target
 */
static int
mptsas_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
{
        mptsas_t                *mpt = ADDR2MPT(ap);
        int                     rval;
        mptsas_tgt_private_t    *tgt_private;
        int                     target, lun;

        tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
            tran_tgt_private;
        ASSERT(tgt_private != NULL);
        target = tgt_private->t_private->m_devhdl;
        lun = tgt_private->t_lun;

        NDBG23(("mptsas_scsi_abort: target=%d.%d", target, lun));

        mutex_enter(&mpt->m_mutex);
        rval = mptsas_do_scsi_abort(mpt, target, lun, pkt);
        mutex_exit(&mpt->m_mutex);
        return (rval);
}

static int
mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun, struct scsi_pkt *pkt)
{
        mptsas_cmd_t    *sp = NULL;
        mptsas_slots_t  *slots = mpt->m_active;
        int             rval = FALSE;

        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * Abort the command pkt on the target/lun in ap.  If pkt is
         * NULL, abort all outstanding commands on that target/lun.
         * If you can abort them, return 1, else return 0.
         * Each packet that’s aborted should be sent back to the target
         * driver through the callback routine, with pkt_reason set to
         * CMD_ABORTED.
         *
         * abort cmd pkt on HBA hardware; clean out of outstanding
         * command lists, etc.
         */
        if (pkt != NULL) {
                /* abort the specified packet */
                sp = PKT2CMD(pkt);

                if (sp->cmd_queued) {
                        NDBG23(("mptsas_do_scsi_abort: queued sp=0x%p aborted",
                            (void *)sp));
                        mptsas_waitq_delete(mpt, sp);
                        mptsas_set_pkt_reason(mpt, sp, CMD_ABORTED,
```

```
                            STAT_ABORTED);
                        mptsas_doneq_add(mpt, sp);
                        rval = TRUE;
                        goto done;
                }

                /*
                 * Have mpt firmware abort this command
                 */

                if (slots->m_slot[sp->cmd_slot] != NULL) {
                        rval = mptsas_ioc_task_management(mpt,
                            MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK, target,
                            lun, NULL, 0, 0);

                        /*
                         * The transport layer expects only TRUE and FALSE.
                         * Therefore, if mptsas_ioc_task_management returns
                         * FAILED we will return FALSE.
                         */
                        if (rval == FAILED)
                                rval = FALSE;
                        goto done;
                }
        }

        /*
         * If pkt is NULL then abort task set
         */
        rval = mptsas_ioc_task_management(mpt,
            MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET, target, lun, NULL, 0, 0);

        /*
         * The transport layer expects only TRUE and FALSE.
         * Therefore, if mptsas_ioc_task_management returns
         * FAILED we will return FALSE.
         */
        if (rval == FAILED)
                rval = FALSE;

#ifdef MPTSAS_TEST
        if (rval && mptsas_test_stop) {
                debug_enter("mptsas_do_scsi_abort");
        }
#endif

done:
        mptsas_doneq_empty(mpt);
        return (rval);
}

/*
 * capability handling:
 * (*tran_getcap).  Get the capability named, and return its value.
 */
static int
mptsas_scsi_getcap(struct scsi_address *ap, char *cap, int tgtonly)
{
        mptsas_t        *mpt = ADDR2MPT(ap);
        int             ckey;
        int             rval = FALSE;

        NDBG24(("mptsas_scsi_getcap: target=%d, cap=%s tgtonly=%x",
            ap->a_target, cap, tgtonly));

        mutex_enter(&mpt->m_mutex);
	if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
		mutex_exit(&mpt->m_mutex);
		return (UNDEFINED);
	}

	switch (ckey) {
	case SCSI_CAP_DMA_MAX:
		rval = (int)mpt->m_msg_dma_attr.dma_attr_maxxfer;
		break;
	case SCSI_CAP_ARQ:
		rval = TRUE;
		break;
	case SCSI_CAP_MSG_OUT:
	case SCSI_CAP_PARITY:
	case SCSI_CAP_UNTAGGED_QING:
		rval = TRUE;
		break;
	case SCSI_CAP_TAGGED_QING:
		rval = TRUE;
		break;
	case SCSI_CAP_RESET_NOTIFICATION:
		rval = TRUE;
		break;
	case SCSI_CAP_LINKED_CMDS:
		rval = FALSE;
		break;
	case SCSI_CAP_QFULL_RETRIES:
		rval = ((mptsas_tgt_private_t *)(ap->a_hba_tran->
		    tran_tgt_private))->t_private->m_qfull_retries;
		break;
	case SCSI_CAP_QFULL_RETRY_INTERVAL:
		rval = drv_hztousec(((mptsas_tgt_private_t *)
		    (ap->a_hba_tran->tran_tgt_private))->
		    t_private->m_qfull_retry_interval) / 1000;
		break;
	case SCSI_CAP_CDB_LEN:
		rval = CDB_GROUP4;
		break;
	case SCSI_CAP_INTERCONNECT_TYPE:
		rval = INTERCONNECT_SAS;
		break;
	case SCSI_CAP_TRAN_LAYER_RETRIES:
		if (mpt->m_ioc_capabilities &
		    MPI2_IOCFACTS_CAPABILITY_TLR)
			rval = TRUE;
		else
			rval = FALSE;
		break;
	default:
		rval = UNDEFINED;
		break;
	}

	NDBG24(("mptsas_scsi_getcap: %s, rval=%x", cap, rval));

	mutex_exit(&mpt->m_mutex);
	return (rval);
}

/*
 * (*tran_setcap).  Set the capability named to the value given.
 */
static int
mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value, int tgtonly)
{
```

```
	mptsas_t	*mpt = ADDR2MPT(ap);
	int		ckey;
	int		rval = FALSE;

	NDBG24(("mptsas_scsi_setcap: target=%d, cap=%s value=%x tgtonly=%x",
	    ap->a_target, cap, value, tgtonly));

	if (!tgtonly) {
		return (rval);
	}

	mutex_enter(&mpt->m_mutex);

	if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
		mutex_exit(&mpt->m_mutex);
		return (UNDEFINED);
	}

	switch (ckey) {
	case SCSI_CAP_DMA_MAX:
	case SCSI_CAP_MSG_OUT:
	case SCSI_CAP_PARITY:
	case SCSI_CAP_INITIATOR_ID:
	case SCSI_CAP_LINKED_CMDS:
	case SCSI_CAP_UNTAGGED_QING:
	case SCSI_CAP_RESET_NOTIFICATION:
		/*
		 * None of these are settable via
		 * the capability interface.
		 */
		break;
	case SCSI_CAP_ARQ:
		/*
		 * We cannot turn off arq so return false if asked to
		 */
		if (value) {
			rval = TRUE;
		} else {
			rval = FALSE;
		}
		break;
	case SCSI_CAP_TAGGED_QING:
		mptsas_set_throttle(mpt, ((mptsas_tgt_private_t *)
		    (ap->a_hba_tran->tran_tgt_private))->t_private,
		    MAX_THROTTLE);
		rval = TRUE;
		break;
	case SCSI_CAP_QFULL_RETRIES:
		((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
		    t_private->m_qfull_retries = (uchar_t)value;
		rval = TRUE;
		break;
	case SCSI_CAP_QFULL_RETRY_INTERVAL:
		((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
		    t_private->m_qfull_retry_interval =
		    drv_usectohz(value * 1000);
		rval = TRUE;
		break;
	default:
		rval = UNDEFINED;
		break;
	}
	mutex_exit(&mpt->m_mutex);
	return (rval);
}
```

```
/*
 * Utility routine for mptsas_ifsetcap/ifgetcap
 */
/*ARGSUSED*/
static int
mptsas_scsi_capchk(char *cap, int tgtonly, int *cidxp)
{
        NDBG24(("mptsas_scsi_capchk: cap=%s", cap));

        if (!cap)
                return (FALSE);

        *cidxp = scsi_hba_lookup_capstr(cap);
        return (TRUE);
}

static int
mptsas_alloc_active_slots(mptsas_t *mpt, int flag)
{
        mptsas_slots_t  *old_active = mpt->m_active;
        mptsas_slots_t  *new_active;
        size_t          size;

        /*
         * if there are active commands, then we cannot
         * change size of active slots array.
         */
        ASSERT(mpt->m_ncmds == 0);

        size = MPTSAS_SLOTS_SIZE(mpt);
        new_active = kmem_zalloc(size, flag);
        if (new_active == NULL) {
                NDBG1(("new active alloc failed"));
                return (-1);
        }
        /*
         * Since SMID 0 is reserved and the TM slot is reserved, the
         * number of slots that can be used at any one time is
         * m_max_requests - 2.
         */
        new_active->m_n_normal = (mpt->m_max_requests - 2);
        new_active->m_size = size;
        new_active->m_rotor = 1;
        if (old_active)
                mptsas_free_active_slots(mpt);
        mpt->m_active = new_active;

        return (0);
}

static void
mptsas_free_active_slots(mptsas_t *mpt)
{
        mptsas_slots_t  *active = mpt->m_active;
        size_t          size;

        if (active == NULL)
                return;
        size = active->m_size;
        kmem_free(active, size);
        mpt->m_active = NULL;
}

/*
 * Error logging, printing, and debug print routines.
 */
```

```
static char *mptsas_label = "mpt_sas";

/*PRINTFLIKE3*/
void
mptsas_log(mptsas_t *mpt, int level, char *fmt, ...)
{
        dev_info_t      *dev;
        va_list         ap;

        if (mpt) {
                dev = mpt->m_dip;
        } else {
                dev = 0;
        }

        mutex_enter(&mptsas_log_mutex);

        va_start(ap, fmt);
        (void) vsprintf(mptsas_log_buf, fmt, ap);
        va_end(ap);

        if (level == CE_CONT) {
                scsi_log(dev, mptsas_label, level, "%s\n", mptsas_log_buf);
        } else {
                scsi_log(dev, mptsas_label, level, "%s", mptsas_log_buf);
        }

        mutex_exit(&mptsas_log_mutex);
}

#ifdef MPTSAS_DEBUG
/*
 * Use a circular buffer to log messages to private memory.
 * Increment idx atomically to minimize risk to miss lines.
 * It’s fast and does not hold up the proceedings too much.
 */
static const size_t mptsas_dbglog_linecnt = MPTSAS_DBGLOG_LINECNT;
static const size_t mptsas_dbglog_linelen = MPTSAS_DBGLOG_LINELEN;
static char mptsas_dbglog_bufs[MPTSAS_DBGLOG_LINECNT][MPTSAS_DBGLOG_LINELEN];
static uint32_t mptsas_dbglog_idx = 0;

/*PRINTFLIKE1*/
void
mptsas_debug_log(char *fmt, ...)
{
        va_list         ap;
        uint32_t        idx;

        idx = atomic_inc_32_nv(&mptsas_dbglog_idx) &
            (mptsas_dbglog_linecnt - 1);

        va_start(ap, fmt);
        (void) vsnprintf(mptsas_dbglog_bufs[idx],
            mptsas_dbglog_linelen, fmt, ap);
        va_end(ap);
}

/*PRINTFLIKE1*/
void
mptsas_printf(char *fmt, ...)
{
        dev_info_t      *dev = 0;
        va_list         ap;

        mutex_enter(&mptsas_log_mutex);
        va_start(ap, fmt);
        (void) vsprintf(mptsas_log_buf, fmt, ap);
        va_end(ap);

#ifdef PROM_PRINTF
        prom_printf("%s:\t%s\n", mptsas_label, mptsas_log_buf);
#else
        scsi_log(dev, mptsas_label, CE_CONT, "!%s\n", mptsas_log_buf);
#endif
        mutex_exit(&mptsas_log_mutex);
}
#endif

/*
 * timeout handling
 */
static void
mptsas_watch(void *arg)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(arg))
#endif

        mptsas_t        *mpt;
        uint32_t        doorbell;

        NDBG30(("mptsas_watch"));

        rw_enter(&mptsas_global_rwlock, RW_READER);
        for (mpt = mptsas_head; mpt != (mptsas_t *)NULL; mpt = mpt->m_next) {

                mutex_enter(&mpt->m_mutex);

                /* Skip device if not powered on */
                if (mpt->m_options & MPTSAS_OPT_PM) {
                        if (mpt->m_power_level == PM_LEVEL_D0) {
                                (void) pm_busy_component(mpt->m_dip, 0);
                                mpt->m_busy = 1;
                        } else {
                                mutex_exit(&mpt->m_mutex);
                                continue;
                        }
                }

                /*
                 * Check if controller is in a FAULT state. If so, reset it.
                 */
                doorbell = ddi_get32(mpt->m_datap, &mpt->m_reg->Doorbell);
                if ((doorbell & MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_FAULT) {
                        doorbell &= MPI2_DOORBELL_DATA_MASK;
                        mptsas_log(mpt, CE_WARN, "MPT Firmware Fault, "
                            "code: %04x", doorbell);
                        mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
                        if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
                                mptsas_log(mpt, CE_WARN, "Reset failed"
                                    "after fault was detected");
                        }
                }

                /*
                 * For now, always call mptsas_watchsubr.
                 */
                mptsas_watchsubr(mpt);

                if (mpt->m_options & MPTSAS_OPT_PM) {
                        mpt->m_busy = 0;
```

```
                                (void) pm_idle_component(mpt->m_dip, 0);
                }

                mutex_exit(&mpt->m_mutex);
        }
        rw_exit(&mptsas_global_rwlock);

        mutex_enter(&mptsas_global_mutex);
        if (mptsas_timeouts_enabled)
                mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
        mutex_exit(&mptsas_global_mutex);
}

static void
mptsas_watchsubr(mptsas_t *mpt)
{
        int             i;
        mptsas_cmd_t    *cmd;
        mptsas_target_t *ptgt = NULL;
        hrtime_t        timestamp = gethrtime();

        ASSERT(MUTEX_HELD(&mpt->m_mutex));

        NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

#ifdef MPTSAS_TEST
        if (mptsas_enable_untagged) {
                mptsas_test_untagged++;
        }
#endif

        /*
         * Check for commands stuck in active slot
         * Account for TM requests, which use the last SMID.
         */
        for (i = 0; i <= mpt->m_active->m_n_normal; i++) {
                if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
                        if (cmd->cmd_active_expiration <= timestamp) {
                                if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
                                        /*
                                         * There seems to be a command stuck
                                         * in the active slot.  Drain throttle.
                                         */
                                        mptsas_set_throttle(mpt,
                                            cmd->cmd_tgt_addr,
                                            DRAIN_THROTTLE);
                                } else if (cmd->cmd_flags &
                                    (CFLAG_PASSTHRU | CFLAG_CONFIG |
                                    CFLAG_FW_DIAG)) {
                                        /*
                                         * passthrough command timeout
                                         */
                                        cmd->cmd_flags |= (CFLAG_FINISHED |
                                            CFLAG_TIMEOUT);
                                        cv_broadcast(&mpt->m_passthru_cv);
                                        cv_broadcast(&mpt->m_config_cv);
                                        cv_broadcast(&mpt->m_fw_diag_cv);
                                }
                        }
                }
        }

        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                /*
                 * If we were draining due to a qfull condition,
		 * go back to full throttle.
		 */
		if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
		    (ptgt->m_t_throttle > HOLD_THROTTLE) &&
		    (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
			mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
			mptsas_restart_hba(mpt);
		}

		cmd = TAILQ_LAST(&ptgt->m_active_cmdq, mptsas_active_cmdq);
		if (cmd == NULL)
			continue;

		if (cmd->cmd_active_expiration <= timestamp) {
			/*
			 * Earliest command timeout expired. Drain throttle.
			 */
			mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);

			/*
			 * Check for remaining commands.
			 */
			cmd = TAILQ_FIRST(&ptgt->m_active_cmdq);
			if (cmd->cmd_active_expiration > timestamp) {
				/*
				 * Wait for remaining commands to complete or
				 * time out.
				 */
				NDBG23(("command timed out, pending drain"));
				continue;
			}

			/*
			 * All command timeouts expired.
			 */
			mptsas_log(mpt, CE_NOTE, "Timeout of %d seconds "
			    "expired with %d commands on target %d lun %d.",
			    cmd->cmd_pkt->pkt_time, ptgt->m_t_ncmds,
			    ptgt->m_devhdl, Lun(cmd));

			mptsas_cmd_timeout(mpt, ptgt);
		} else if (cmd->cmd_active_expiration <=
		    timestamp + (hrtime_t)mptsas_scsi_watchdog_tick * NANOSEC) {
			NDBG23(("pending timeout"));
			mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
		}
	}
}

/*
 * timeout recovery
 */
static void
mptsas_cmd_timeout(mptsas_t *mpt, mptsas_target_t *ptgt)
{
	uint16_t	devhdl;
	uint64_t	sas_wwn;
	uint8_t		phy;
	char		wwn_str[MPTSAS_WWN_STRLEN];

	devhdl = ptgt->m_devhdl;
	sas_wwn = ptgt->m_addr.mta_wwn;
	phy = ptgt->m_phynum;
	if (sas_wwn == 0) {
		(void) sprintf(wwn_str, "p%x", phy);
	} else {
```

```
		(void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
	}

	NDBG29(("mptsas_cmd_timeout: target=%d", devhdl));
	mptsas_log(mpt, CE_WARN, "Disconnected command timeout for "
	    "target %d %s, enclosure %u", devhdl, wwn_str,
	    ptgt->m_enclosure);

	/*
	 * Abort all outstanding commands on the device.
	 */
	NDBG29(("mptsas_cmd_timeout: device reset"));
	if (mptsas_do_scsi_reset(mpt, devhdl) != TRUE) {
		mptsas_log(mpt, CE_WARN, "Target %d reset for command timeout "
		    "recovery failed!", devhdl);
	}
}

/*
 * Device / Hotplug control
 */
static int
mptsas_scsi_quiesce(dev_info_t *dip)
{
	mptsas_t	*mpt;
	scsi_hba_tran_t	*tran;

	tran = ddi_get_driver_private(dip);
	if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
		return (-1);

	return (mptsas_quiesce_bus(mpt));
}

static int
mptsas_scsi_unquiesce(dev_info_t *dip)
{
	mptsas_t		*mpt;
	scsi_hba_tran_t	*tran;

	tran = ddi_get_driver_private(dip);
	if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
		return (-1);

	return (mptsas_unquiesce_bus(mpt));
}

static int
mptsas_quiesce_bus(mptsas_t *mpt)
{
	mptsas_target_t	*ptgt = NULL;

	NDBG28(("mptsas_quiesce_bus"));
	mutex_enter(&mpt->m_mutex);

	/* Set all the throttles to zero */
	for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
	    ptgt = refhash_next(mpt->m_targets, ptgt)) {
		mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
	}

	/* If there are any outstanding commands in the queue */
	if (mpt->m_ncmds) {
		mpt->m_softstate |= MPTSAS_SS_DRAINING;
		mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
		    mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
```

```
                if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
                        /*
                         * Quiesce has been interrupted
                         */
                        mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
                        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
                            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
                        }
                        mptsas_restart_hba(mpt);
                        if (mpt->m_quiesce_timeid != 0) {
                                timeout_id_t tid = mpt->m_quiesce_timeid;
                                mpt->m_quiesce_timeid = 0;
                                mutex_exit(&mpt->m_mutex);
                                (void) untimeout(tid);
                                return (-1);
                        }
                        mutex_exit(&mpt->m_mutex);
                        return (-1);
                } else {
                        /* Bus has been quiesced */
                        ASSERT(mpt->m_quiesce_timeid == 0);
                        mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
                        mpt->m_softstate |= MPTSAS_SS_QUIESCED;
                        mutex_exit(&mpt->m_mutex);
                        return (0);
                }
        }
        /* Bus was not busy - QUIESCED */
        mutex_exit(&mpt->m_mutex);

        return (0);
}

static int
mptsas_unquiesce_bus(mptsas_t *mpt)
{
        mptsas_target_t *ptgt = NULL;

        NDBG28(("mptsas_unquiesce_bus"));
        mutex_enter(&mpt->m_mutex);
        mpt->m_softstate &= ~MPTSAS_SS_QUIESCED;
        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
        }
        mptsas_restart_hba(mpt);
        mutex_exit(&mpt->m_mutex);
        return (0);
}

static void
mptsas_ncmds_checkdrain(void *arg)
{
        mptsas_t        *mpt = arg;
        mptsas_target_t *ptgt = NULL;

        mutex_enter(&mpt->m_mutex);
        if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
                mpt->m_quiesce_timeid = 0;
                if (mpt->m_ncmds == 0) {
                        /* Command queue has been drained */
                        cv_signal(&mpt->m_cv);
                } else {
                        /*
                         * The throttle may have been reset because
```

```
                         * of a SCSI bus reset
                         */
                        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
                            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                                mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
                        }

                        mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
                            mpt, (MPTSAS_QUIESCE_TIMEOUT *
                            drv_usectohz(1000000)));
                }
        }
        mutex_exit(&mpt->m_mutex);
}

/*ARGSUSED*/
static void
mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        int     i;
        uint8_t *cp = (uchar_t *)cmd->cmd_pkt->pkt_cdbp;
        char    buf[128];

        buf[0] = '\0';
        NDBG25(("?Cmd (0x%p) dump for Target %d Lun %d:\n", (void *)cmd,
            Tgt(cmd), Lun(cmd)));
        (void) sprintf(&buf[0], "\tcdb=[");
        for (i = 0; i < (int)cmd->cmd_cdblen; i++) {
                (void) sprintf(&buf[strlen(buf)], " 0x%x", *cp++);
        }
        (void) sprintf(&buf[strlen(buf)], " ]");
        NDBG25(("?%s\n", buf));
        NDBG25(("?pkt_flags=0x%x pkt_statistics=0x%x pkt_state=0x%x\n",
            cmd->cmd_pkt->pkt_flags, cmd->cmd_pkt->pkt_statistics,
            cmd->cmd_pkt->pkt_state));
        NDBG25(("?pkt_scbp=0x%x cmd_flags=0x%x\n", cmd->cmd_pkt->pkt_scbp ?
            *(cmd->cmd_pkt->pkt_scbp) : 0, cmd->cmd_flags));
}

static void
mptsas_passthru_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
    pMpi2SGESimple64_t sgep)
{
        uint32_t                sge_flags;
        uint32_t                data_size, dataout_size;
        ddi_dma_cookie_t        data_cookie;
        ddi_dma_cookie_t        dataout_cookie;

        data_size = pt->data_size;
        dataout_size = pt->dataout_size;
        data_cookie = pt->data_cookie;
        dataout_cookie = pt->dataout_cookie;

        if (dataout_size) {
                sge_flags = dataout_size |
                    ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
                    MPI2_SGE_FLAGS_END_OF_BUFFER |
                    MPI2_SGE_FLAGS_HOST_TO_IOC |
                    MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
                    MPI2_SGE_FLAGS_SHIFT);
                ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
                ddi_put32(acc_hdl, &sgep->Address.Low,
                    (uint32_t)(dataout_cookie.dmac_laddress &
                    0xffffffffull));
                ddi_put32(acc_hdl, &sgep->Address.High,
                    (uint32_t)(dataout_cookie.dmac_laddress
```

```c
                    >> 32));
                sgep++;
        }
        sge_flags = data_size;
        sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
            MPI2_SGE_FLAGS_LAST_ELEMENT |
            MPI2_SGE_FLAGS_END_OF_BUFFER |
            MPI2_SGE_FLAGS_END_OF_LIST |
            MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
            MPI2_SGE_FLAGS_SHIFT);
        if (pt->direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
                sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
                    MPI2_SGE_FLAGS_SHIFT);
        } else {
                sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
                    MPI2_SGE_FLAGS_SHIFT);
        }
        ddi_put32(acc_hdl, &sgep->FlagsLength,
            sge_flags);
        ddi_put32(acc_hdl, &sgep->Address.Low,
            (uint32_t)(data_cookie.dmac_laddress &
            0xffffffffull));
        ddi_put32(acc_hdl, &sgep->Address.High,
            (uint32_t)(data_cookie.dmac_laddress >> 32));
}

static void
mptsas_passthru_ieee_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
    pMpi2IeeeSgeSimple64_t ieeesgep)
{
        uint8_t                 sge_flags;
        uint32_t                data_size, dataout_size;
        ddi_dma_cookie_t        data_cookie;
        ddi_dma_cookie_t        dataout_cookie;

        data_size = pt->data_size;
        dataout_size = pt->dataout_size;
        data_cookie = pt->data_cookie;
        dataout_cookie = pt->dataout_cookie;

        sge_flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
            MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
        if (dataout_size) {
                ddi_put32(acc_hdl, &ieeesgep->Length, dataout_size);
                ddi_put32(acc_hdl, &ieeesgep->Address.Low,
                    (uint32_t)(dataout_cookie.dmac_laddress &
                    0xffffffffull));
                ddi_put32(acc_hdl, &ieeesgep->Address.High,
                    (uint32_t)(dataout_cookie.dmac_laddress >> 32));
                ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
                ieeesgep++;
        }
        sge_flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
        ddi_put32(acc_hdl, &ieeesgep->Length, data_size);
        ddi_put32(acc_hdl, &ieeesgep->Address.Low,
            (uint32_t)(data_cookie.dmac_laddress & 0xffffffffull));
        ddi_put32(acc_hdl, &ieeesgep->Address.High,
            (uint32_t)(data_cookie.dmac_laddress >> 32));
        ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
}

static void
mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        caddr_t                 memp;
        pMPI2RequestHeader_t    request_hdrp;
```

```c
        struct scsi_pkt         *pkt = cmd->cmd_pkt;
        mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
        uint32_t                request_size;
        uint32_t                request_desc_low, request_desc_high = 0;
        uint32_t                i, sense_bufp;
        uint8_t                 desc_type;
        uint8_t                 *request, function;
        ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
        ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

        desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

        request = pt->request;
        request_size = pt->request_size;

        /*
         * Store the passthrough message in memory location
         * corresponding to our slot number
         */
        memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
        request_hdrp = (pMPI2RequestHeader_t)memp;
        bzero(memp, mpt->m_req_frame_size);

        for (i = 0; i < request_size; i++) {
                bcopy(request + i, memp + i, 1);
        }

        NDBG15(("mptsas_start_passthru: Func 0x%x, MsgFlags 0x%x, "
            "size=%d, in %d, out %d", request_hdrp->Function,
            request_hdrp->MsgFlags, request_size,
            pt->data_size, pt->dataout_size));

        /*
         * Add an SGE, even if the length is zero.
         */
        if (mpt->m_MPI25 && pt->simple == 0) {
                mptsas_passthru_ieee_sge(acc_hdl, pt,
                    (pMpi2IeeeSgeSimple64_t)
                    ((uint8_t *)request_hdrp + pt->sgl_offset));
        } else {
                mptsas_passthru_sge(acc_hdl, pt,
                    (pMpi2SGESimple64_t)
                    ((uint8_t *)request_hdrp + pt->sgl_offset));
        }

        function = request_hdrp->Function;
        if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
            (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
                pMpi2SCSIIORequest_t    scsi_io_req;

                NDBG15(("mptsas_start_passthru: Is SCSI IO Req"));
                scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;
                /*
                 * Put SGE for data and data_out buffer at the end of
                 * scsi_io_request message header.(64 bytes in total)
                 * Following above SGEs, the residual space will be
                 * used by sense data.
                 */
                ddi_put8(acc_hdl,
                    &scsi_io_req->SenseBufferLength,
                    (uint8_t)(request_size - 64));

                sense_bufp = mpt->m_req_frame_dma_addr +
                    (mpt->m_req_frame_size * cmd->cmd_slot);
                sense_bufp += 64;
                ddi_put32(acc_hdl,
                    &scsi_io_req->SenseBufferLowAddress, sense_bufp);

                /*
                 * Set SGLOffset0 value
                 */
                ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
                    offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);

                /*
                 * Setup descriptor info.  RAID passthrough must use the
                 * default request descriptor which is already set, so if this
                 * is a SCSI IO request, change the descriptor to SCSI IO.
                 */
                if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
                        desc_type = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
                        request_desc_high = (ddi_get16(acc_hdl,
                            &scsi_io_req->DevHandle) << 16);
                }
        }

        /*
         * We must wait till the message has been completed before
         * beginning the next message so we wait for this one to
         * finish.
         */
        (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
        request_desc_low = (cmd->cmd_slot << 16) + desc_type;
        cmd->cmd_rfm = NULL;
        MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
        if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
            (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
        }
}

typedef void (mptsas_pre_f)(mptsas_t *, mptsas_pt_request_t *);
static mptsas_pre_f     mpi_pre_ioc_facts;
static mptsas_pre_f     mpi_pre_port_facts;
static mptsas_pre_f     mpi_pre_fw_download;
static mptsas_pre_f     mpi_pre_fw_25_download;
static mptsas_pre_f     mpi_pre_fw_upload;
static mptsas_pre_f     mpi_pre_fw_25_upload;
static mptsas_pre_f     mpi_pre_sata_passthrough;
static mptsas_pre_f     mpi_pre_smp_passthrough;
static mptsas_pre_f     mpi_pre_config;
static mptsas_pre_f     mpi_pre_sas_io_unit_control;
static mptsas_pre_f     mpi_pre_scsi_io_req;

/*
 * Prepare the pt for a SAS2 FW_DOWNLOAD request.
 */
static void
mpi_pre_fw_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
        pMpi2FWDownloadTCSGE_t tcsge;
        pMpi2FWDownloadRequest req;

        /*
         * If SAS3, call separate function.
         */
        if (mpt->m_MPI25) {
                mpi_pre_fw_25_download(mpt, pt);
                return;
        }

        /*
```

```
         * User requests should come in with the Transaction
         * context element where the SGL will go. Putting the
         * SGL after that seems to work, but don't really know
         * why. Other drivers tend to create an extra SGL and
         * refer to the TCE through that.
         */
        req = (pMpi2FWDownloadRequest)pt->request;
        tcsge = (pMpi2FWDownloadTCSGE_t)&req->SGL;
        if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
            tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
                mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
        }

        pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL) +
            sizeof (*tcsge);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_fw_download(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    (int)pt->request_size, (int)pt->sgl_offset,
                    (int)pt->dataout_size));
        if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
                NDBG15(("mpi_pre_fw_download(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_FW_DOWNLOAD_REPLY)));
}

/*
 * Prepare the pt for a SAS3 FW_DOWNLOAD request.
 */
static void
mpi_pre_fw_25_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
        pMpi2FWDownloadTCSGE_t tcsge;
        pMpi2FWDownloadRequest req2;
        pMpi25FWDownloadRequest req25;

        /*
         * User requests should come in with the Transaction
         * context element where the SGL will go. The new firmware
         * Doesn't use TCE and has space in the main request for
         * this information. So move to the right place.
         */
        req2 = (pMpi2FWDownloadRequest)pt->request;
        req25 = (pMpi25FWDownloadRequest)pt->request;
        tcsge = (pMpi2FWDownloadTCSGE_t)&req2->SGL;
        if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
            tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
                mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
        }
        req25->ImageOffset = tcsge->ImageOffset;
        req25->ImageSize = tcsge->ImageSize;

        pt->sgl_offset = offsetof(MPI25_FW_DOWNLOAD_REQUEST, SGL);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_fw_25_download(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size, pt->sgl_offset,
                    pt->dataout_size));
        if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
                NDBG15(("mpi_pre_fw_25_download(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
}

/*
 * Prepare the pt for a SAS2 FW_UPLOAD request.
```

```c
 */
static void
mpi_pre_fw_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
        pMpi2FWUploadTCSGE_t tcsge;
        pMpi2FWUploadRequest_t req;

        /*
         * If SAS3, call separate function.
         */
        if (mpt->m_MPI25) {
                mpi_pre_fw_25_upload(mpt, pt);
                return;
        }

        /*
         * User requests should come in with the Transaction
         * context element where the SGL will go. Putting the
         * SGL after that seems to work, but don't really know
         * why. Other drivers tend to create an extra SGL and
         * refer to the TCE through that.
         */
        req = (pMpi2FWUploadRequest_t)pt->request;
        tcsge = (pMpi2FWUploadTCSGE_t)&req->SGL;
        if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
            tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
                mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
        }

        pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL) +
            sizeof (*tcsge);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_fw_upload(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size, pt->sgl_offset,
                    pt->dataout_size));
        if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
                NDBG15(("mpi_pre_fw_upload(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
}

/*
 * Prepare the pt a SAS3 FW_UPLOAD request.
 */
static void
mpi_pre_fw_25_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
        pMpi2FWUploadTCSGE_t tcsge;
        pMpi2FWUploadRequest_t req2;
        pMpi25FWUploadRequest_t req25;

        /*
         * User requests should come in with the Transaction
         * context element where the SGL will go. The new firmware
         * Doesn't use TCE and has space in the main request for
         * this information. So move to the right place.
         */
        req2 = (pMpi2FWUploadRequest_t)pt->request;
        req25 = (pMpi25FWUploadRequest_t)pt->request;
        tcsge = (pMpi2FWUploadTCSGE_t)&req2->SGL;
        if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
            tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
                mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
        }
        req25->ImageOffset = tcsge->ImageOffset;
```

```c
        req25->ImageSize = tcsge->ImageSize;

        pt->sgl_offset = offsetof(MPI25_FW_UPLOAD_REQUEST, SGL);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_fw_25_upload(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size, pt->sgl_offset,
                    pt->dataout_size));
        if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
                NDBG15(("mpi_pre_fw_25_upload(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
}

/*
 * Prepare the pt for an IOC_FACTS request.
 */
static void
mpi_pre_ioc_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        if (pt->request_size != sizeof (MPI2_IOC_FACTS_REQUEST))
                NDBG15(("mpi_pre_ioc_facts(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size,
                    (int)sizeof (MPI2_IOC_FACTS_REQUEST),
                    pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_IOC_FACTS_REPLY))
                NDBG15(("mpi_pre_ioc_facts(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_IOC_FACTS_REPLY)));
        pt->sgl_offset = (uint16_t)pt->request_size;
}

/*
 * Prepare the pt for a PORT_FACTS request.
 */
static void
mpi_pre_port_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        if (pt->request_size != sizeof (MPI2_PORT_FACTS_REQUEST))
                NDBG15(("mpi_pre_port_facts(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size,
                    (int)sizeof (MPI2_PORT_FACTS_REQUEST),
                    pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_PORT_FACTS_REPLY))
                NDBG15(("mpi_pre_port_facts(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_PORT_FACTS_REPLY)));
        pt->sgl_offset = (uint16_t)pt->request_size;
}

/*
 * Prepare pt for a SATA_PASSTHROUGH request.
 */
static void
mpi_pre_sata_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
```

```
#endif
        pt->sgl_offset = offsetof(MPI2_SATA_PASSTHROUGH_REQUEST, SGL);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_sata_passthrough(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size, pt->sgl_offset,
                    pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_SATA_PASSTHROUGH_REPLY))
                NDBG15(("mpi_pre_sata_passthrough(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_SATA_PASSTHROUGH_REPLY)));
}

static void
mpi_pre_smp_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        pt->sgl_offset = offsetof(MPI2_SMP_PASSTHROUGH_REQUEST, SGL);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_smp_passthrough(): Incorrect req size, "
                    "0x%x, should be 0x%x, dataoutsz 0x%x",
                    pt->request_size, pt->sgl_offset,
                    pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_SMP_PASSTHROUGH_REPLY))
                NDBG15(("mpi_pre_smp_passthrough(): Incorrect rep size, "
                    "0x%x, should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_SMP_PASSTHROUGH_REPLY)));
}

/*
 * Prepare pt for a CONFIG request.
 */
static void
mpi_pre_config(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        pt->sgl_offset = offsetof(MPI2_CONFIG_REQUEST, PageBufferSGE);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
                    "should be 0x%x, dataoutsz 0x%x", pt->request_size,
                    pt->sgl_offset, pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_CONFIG_REPLY))
                NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
                    "should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_CONFIG_REPLY)));
        pt->simple = 1;
}

/*
 * Prepare pt for a SCSI_IO_REQ request.
 */
static void
mpi_pre_scsi_io_req(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        pt->sgl_offset = offsetof(MPI2_SCSI_IO_REQUEST, SGL);
        if (pt->request_size != pt->sgl_offset)
                NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
                    "should be 0x%x, dataoutsz 0x%x", pt->request_size,
                    pt->sgl_offset,
```

```
                    pt->dataout_size));
        if (pt->data_size != sizeof (MPI2_SCSI_IO_REPLY))
                NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
                    "should be 0x%x", pt->data_size,
                    (int)sizeof (MPI2_SCSI_IO_REPLY)));
}

/*
 * Prepare the mptsas_cmd for a SAS_IO_UNIT_CONTROL request.
 */
static void
mpi_pre_sas_io_unit_control(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(mpt))
#endif
        pt->sgl_offset = (uint16_t)pt->request_size;
}

/*
 * A set of functions to prepare an mptsas_cmd for the various
 * supported requests.
 */
static struct mptsas_func {
        U8              Function;
        char            *Name;
        mptsas_pre_f    *f_pre;
} mptsas_func_list[] = {
        { MPI2_FUNCTION_IOC_FACTS, "IOC_FACTS",         mpi_pre_ioc_facts },
        { MPI2_FUNCTION_PORT_FACTS, "PORT_FACTS",       mpi_pre_port_facts },
        { MPI2_FUNCTION_FW_DOWNLOAD, "FW_DOWNLOAD",     mpi_pre_fw_download },
        { MPI2_FUNCTION_FW_UPLOAD, "FW_UPLOAD",         mpi_pre_fw_upload },
        { MPI2_FUNCTION_SATA_PASSTHROUGH, "SATA_PASSTHROUGH",
            mpi_pre_sata_passthrough },
        { MPI2_FUNCTION_SMP_PASSTHROUGH, "SMP_PASSTHROUGH",
            mpi_pre_smp_passthrough},
        { MPI2_FUNCTION_SCSI_IO_REQUEST, "SCSI_IO_REQUEST",
            mpi_pre_scsi_io_req},
        { MPI2_FUNCTION_CONFIG, "CONFIG",               mpi_pre_config},
        { MPI2_FUNCTION_SAS_IO_UNIT_CONTROL, "SAS_IO_UNIT_CONTROL",
            mpi_pre_sas_io_unit_control },
        { 0xFF, NULL,                           NULL } /* list end */
};

static void
mptsas_prep_sgl_offset(mptsas_t *mpt, mptsas_pt_request_t *pt)
{
        pMPI2RequestHeader_t    hdr;
        struct mptsas_func      *f;

        hdr = (pMPI2RequestHeader_t)pt->request;

        for (f = mptsas_func_list; f->f_pre != NULL; f++) {
                if (hdr->Function == f->Function) {
                        f->f_pre(mpt, pt);
                        NDBG15(("mptsas_prep_sgl_offset: Function %s,"
                            " sgl_offset 0x%x", f->Name,
                            pt->sgl_offset));
                        return;
                }
        }
        NDBG15(("mptsas_prep_sgl_offset: Unknown Function 0x%02x,"
            " returning req_size 0x%x for sgl_offset",
            hdr->Function, pt->request_size));
        pt->sgl_offset = (uint16_t)pt->request_size;
}
```

```
static int
mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
    uint8_t *data, uint32_t request_size, uint32_t reply_size,
    uint32_t data_size, uint32_t direction, uint8_t *dataout,
    uint32_t dataout_size, short timeout, int mode)
{
        mptsas_pt_request_t             pt;
        mptsas_dma_alloc_state_t        data_dma_state;
        mptsas_dma_alloc_state_t        dataout_dma_state;
        caddr_t                         memp;
        mptsas_cmd_t                    *cmd = NULL;
        struct scsi_pkt                 *pkt;
        uint32_t                        reply_len = 0, sense_len = 0;
        pMPI2RequestHeader_t            request_hdrp;
        pMPI2RequestHeader_t            request_msg;
        pMPI2DefaultReply_t             reply_msg;
        Mpi2SCSIIOReply_t               rep_msg;
        int                             i, status = 0, pt_flags = 0, rv = 0;
        int                             rvalue;
        uint8_t                         function;

        ASSERT(mutex_owned(&mpt->m_mutex));

        reply_msg = (pMPI2DefaultReply_t)(&rep_msg);
        bzero(reply_msg, sizeof (MPI2_DEFAULT_REPLY));
        request_msg = kmem_zalloc(request_size, KM_SLEEP);

        mutex_exit(&mpt->m_mutex);
        /*
         * copy in the request buffer since it could be used by
         * another thread when the pt request into waitq
         */
        if (ddi_copyin(request, request_msg, request_size, mode)) {
                mutex_enter(&mpt->m_mutex);
                status = EFAULT;
                mptsas_log(mpt, CE_WARN, "failed to copy request data");
                goto out;
        }
        mutex_enter(&mpt->m_mutex);

        function = request_msg->Function;
        if (function == MPI2_FUNCTION_SCSI_TASK_MGMT) {
                pMpi2SCSITaskManagementRequest_t        task;
                task = (pMpi2SCSITaskManagementRequest_t)request_msg;
                mptsas_setup_bus_reset_delay(mpt);
                rv = mptsas_ioc_task_management(mpt, task->TaskType,
                    task->DevHandle, (int)task->LUN[1], reply, reply_size,
                    mode);

                if (rv != TRUE) {
                        status = EIO;
                        mptsas_log(mpt, CE_WARN, "task management failed");
                }
                goto out;
        }

        if (data_size != 0) {
                data_dma_state.size = data_size;
                if (mptsas_dma_alloc(mpt, &data_dma_state) != DDI_SUCCESS) {
                        status = ENOMEM;
                        mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
                            "resource");
                        goto out;
                }
```

```
                pt_flags |= MPTSAS_DATA_ALLOCATED;
                if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
                        mutex_exit(&mpt->m_mutex);
                        for (i = 0; i < data_size; i++) {
                                if (ddi_copyin(data + i, (uint8_t *)
                                    data_dma_state.memp + i, 1, mode)) {
                                        mutex_enter(&mpt->m_mutex);
                                        status = EFAULT;
                                        mptsas_log(mpt, CE_WARN, "failed to "
                                            "copy read data");
                                        goto out;
                                }
                        }
                        mutex_enter(&mpt->m_mutex);
                }
        } else {
                bzero(&data_dma_state, sizeof (data_dma_state));
        }

        if (dataout_size != 0) {
                dataout_dma_state.size = dataout_size;
                if (mptsas_dma_alloc(mpt, &dataout_dma_state) != DDI_SUCCESS) {
                        status = ENOMEM;
                        mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
                            "resource");
                        goto out;
                }
                pt_flags |= MPTSAS_DATAOUT_ALLOCATED;
                mutex_exit(&mpt->m_mutex);
                for (i = 0; i < dataout_size; i++) {
                        if (ddi_copyin(dataout + i, (uint8_t *)
                            dataout_dma_state.memp + i, 1, mode)) {
                                mutex_enter(&mpt->m_mutex);
                                mptsas_log(mpt, CE_WARN, "failed to copy out"
                                    " data");
                                status = EFAULT;
                                goto out;
                        }
                }
                mutex_enter(&mpt->m_mutex);
        } else {
                bzero(&dataout_dma_state, sizeof (dataout_dma_state));
        }

        if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
                status = EAGAIN;
                mptsas_log(mpt, CE_NOTE, "event ack command pool is full");
                goto out;
        }
        pt_flags |= MPTSAS_REQUEST_POOL_CMD;

        bzero((caddr_t)cmd, sizeof (*cmd));
        bzero((caddr_t)pkt, scsi_pkt_size());
        bzero((caddr_t)&pt, sizeof (pt));

        cmd->ioc_cmd_slot = (uint32_t)(rvalue);

        pt.request = (uint8_t *)request_msg;
        pt.direction = direction;
        pt.simple = 0;
        pt.request_size = request_size;
        pt.data_size = data_size;
        pt.dataout_size = dataout_size;
        pt.data_cookie = data_dma_state.cookie;
        pt.dataout_cookie = dataout_dma_state.cookie;
        mptsas_prep_sgl_offset(mpt, &pt);
        /*
         * Form a blank cmd/pkt to store the acknowledgement message
         */
        pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
        pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
        pkt->pkt_ha_private     = (opaque_t)&pt;
        pkt->pkt_flags          = FLAG_HEAD;
        pkt->pkt_time           = timeout;
        cmd->cmd_pkt            = pkt;
        cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_PASSTHRU;

        /*
         * Save the command in a slot
         */
        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                /*
                 * Once passthru command get slot, set cmd_flags
                 * CFLAG_PREPARED.
                 */
                cmd->cmd_flags |= CFLAG_PREPARED;
                mptsas_start_passthru(mpt, cmd);
        } else {
                mptsas_waitq_add(mpt, cmd);
        }

        while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
                cv_wait(&mpt->m_passthru_cv, &mpt->m_mutex);
        }

        if (cmd->cmd_flags & CFLAG_PREPARED) {
                memp = mpt->m_req_frame + (mpt->m_req_frame_size *
                    cmd->cmd_slot);
                request_hdrp = (pMPI2RequestHeader_t)memp;
        }

        if (cmd->cmd_flags & CFLAG_TIMEOUT) {
                status = ETIMEDOUT;
                mptsas_log(mpt, CE_WARN, "passthrough command timeout");
                pt_flags |= MPTSAS_CMD_TIMEOUT;
                goto out;
        }

        if (cmd->cmd_rfm) {
                /*
                 * cmd_rfm is zero means the command reply is a CONTEXT
                 * reply and no PCI Write to post the free reply SMFA
                 * because no reply message frame is used.
                 * cmd_rfm is non-zero means the reply is a ADDRESS
                 * reply and reply message frame is used.
                 */
                pt_flags |= MPTSAS_ADDRESS_REPLY;
                (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
                    DDI_DMA_SYNC_FORCPU);
                reply_msg = (pMPI2DefaultReply_t)
                    (mpt->m_reply_frame + (cmd->cmd_rfm -
                    mpt->m_reply_frame_dma_addr));
        }

        mptsas_fma_check(mpt, cmd);
        if (pkt->pkt_reason == CMD_TRAN_ERR) {
                status = EAGAIN;
                mptsas_log(mpt, CE_WARN, "passthru fma error");
                goto out;
        }
        if (pkt->pkt_reason == CMD_RESET) {
```

```
                status = EAGAIN;
                mptsas_log(mpt, CE_WARN, "ioc reset abort passthru");
                goto out;
        }

        if (pkt->pkt_reason == CMD_INCOMPLETE) {
                status = EIO;
                mptsas_log(mpt, CE_WARN, "passthrough command incomplete");
                goto out;
        }

        mutex_exit(&mpt->m_mutex);
        if (cmd->cmd_flags & CFLAG_PREPARED) {
                function = request_hdrp->Function;
                if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
                    (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
                        reply_len = sizeof (MPI2_SCSI_IO_REPLY);
                        sense_len = reply_size - reply_len;
                } else {
                        reply_len = reply_size;
                        sense_len = 0;
                }

                for (i = 0; i < reply_len; i++) {
                        if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
                            mode)) {
                                mutex_enter(&mpt->m_mutex);
                                status = EFAULT;
                                mptsas_log(mpt, CE_WARN, "failed to copy out "
                                    "reply data");
                                goto out;
                        }
                }
                for (i = 0; i < sense_len; i++) {
                        if (ddi_copyout((uint8_t *)request_hdrp + 64 + i,
                            reply + reply_len + i, 1, mode)) {
                                mutex_enter(&mpt->m_mutex);
                                status = EFAULT;
                                mptsas_log(mpt, CE_WARN, "failed to copy out "
                                    "sense data");
                                goto out;
                        }
                }
        }

        if (data_size) {
                if (direction != MPTSAS_PASS_THRU_DIRECTION_WRITE) {
                        (void) ddi_dma_sync(data_dma_state.handle, 0, 0,
                            DDI_DMA_SYNC_FORCPU);
                        for (i = 0; i < data_size; i++) {
                                if (ddi_copyout((uint8_t *)(
                                    data_dma_state.memp + i), data + i,  1,
                                    mode)) {
                                        mutex_enter(&mpt->m_mutex);
                                        status = EFAULT;
                                        mptsas_log(mpt, CE_WARN, "failed to "
                                            "copy out the reply data");
                                        goto out;
                                }
                        }
                }
        }
        mutex_enter(&mpt->m_mutex);
out:
        /*
         * Put the reply frame back on the free queue, increment the free
         * index, and write the new index to the free index register.  But only
         * if this reply is an ADDRESS reply.
         */
        if (pt_flags & MPTSAS_ADDRESS_REPLY) {
                ddi_put32(mpt->m_acc_free_queue_hdl,
                    &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
                    cmd->cmd_rfm);
                (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
                    DDI_DMA_SYNC_FORDEV);
                if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                        mpt->m_free_index = 0;
                }
                ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
                    mpt->m_free_index);
        }
        if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
                mptsas_remove_cmd(mpt, cmd);
                pt_flags &= (~MPTSAS_REQUEST_POOL_CMD);
        }
        if (pt_flags & MPTSAS_REQUEST_POOL_CMD)
                mptsas_return_to_pool(mpt, cmd);
        if (pt_flags & MPTSAS_DATA_ALLOCATED) {
                if (mptsas_check_dma_handle(data_dma_state.handle) !=
                    DDI_SUCCESS) {
                        ddi_fm_service_impact(mpt->m_dip,
                            DDI_SERVICE_UNAFFECTED);
                        status = EFAULT;
                }
                mptsas_dma_free(&data_dma_state);
        }
        if (pt_flags & MPTSAS_DATAOUT_ALLOCATED) {
                if (mptsas_check_dma_handle(dataout_dma_state.handle) !=
                    DDI_SUCCESS) {
                        ddi_fm_service_impact(mpt->m_dip,
                            DDI_SERVICE_UNAFFECTED);
                        status = EFAULT;
                }
                mptsas_dma_free(&dataout_dma_state);
        }
        if (pt_flags & MPTSAS_CMD_TIMEOUT) {
                if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
                        mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
                }
        }
        if (request_msg)
                kmem_free(request_msg, request_size);

        return (status);
}

static int
mptsas_pass_thru(mptsas_t *mpt, mptsas_pass_thru_t *data, int mode)
{
        /*
         * If timeout is 0, set timeout to default of 60 seconds.
         */
        if (data->Timeout == 0) {
                data->Timeout = MPTSAS_PASS_THRU_TIME_DEFAULT;
        }

        if (((data->DataSize == 0) &&
            (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_NONE)) ||
            ((data->DataSize != 0) &&
            ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_READ) ||
            (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_WRITE) ||
            ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) &&
```

```
            (data->DataOutSize != 0))))) {
                if (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) {
                        data->DataDirection = MPTSAS_PASS_THRU_DIRECTION_READ;
                } else {
                        data->DataOutSize = 0;
                }
                /*
                 * Send passthru request messages
                 */
                return (mptsas_do_passthru(mpt,
                    (uint8_t *)((uintptr_t)data->PtrRequest),
                    (uint8_t *)((uintptr_t)data->PtrReply),
                    (uint8_t *)((uintptr_t)data->PtrData),
                    data->RequestSize, data->ReplySize,
                    data->DataSize, data->DataDirection,
                    (uint8_t *)((uintptr_t)data->PtrDataOut),
                    data->DataOutSize, data->Timeout, mode));
        } else {
                return (EINVAL);
        }
}

static uint8_t
mptsas_get_fw_diag_buffer_number(mptsas_t *mpt, uint32_t unique_id)
{
        uint8_t index;

        for (index = 0; index < MPI2_DIAG_BUF_TYPE_COUNT; index++) {
                if (mpt->m_fw_diag_buffer_list[index].unique_id == unique_id) {
                        return (index);
                }
        }

        return (MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND);
}

static void
mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        pMpi2DiagBufferPostRequest_t    pDiag_post_msg;
        pMpi2DiagReleaseRequest_t       pDiag_release_msg;
        struct scsi_pkt                 *pkt = cmd->cmd_pkt;
        mptsas_diag_request_t           *diag = pkt->pkt_ha_private;
        uint32_t                        request_desc_low, i;

        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * Form the diag message depending on the post or release function.
         */
        if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
                pDiag_post_msg = (pMpi2DiagBufferPostRequest_t)
                    (mpt->m_req_frame + (mpt->m_req_frame_size *
                    cmd->cmd_slot));
                bzero(pDiag_post_msg, mpt->m_req_frame_size);
                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
                    diag->function);
                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
                    diag->pBuffer->buffer_type);
                ddi_put8(mpt->m_acc_req_frame_hdl,
                    &pDiag_post_msg->ExtendedType,
                    diag->pBuffer->extended_type);
                ddi_put32(mpt->m_acc_req_frame_hdl,
                    &pDiag_post_msg->BufferLength,
                    diag->pBuffer->buffer_data.size);
                for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);
                    i++) {
                        ddi_put32(mpt->m_acc_req_frame_hdl,
                            &pDiag_post_msg->ProductSpecific[i],
                            diag->pBuffer->product_specific[i]);
                }
                ddi_put32(mpt->m_acc_req_frame_hdl,
                    &pDiag_post_msg->BufferAddress.Low,
                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
                    & 0xffffffffull));
                ddi_put32(mpt->m_acc_req_frame_hdl,
                    &pDiag_post_msg->BufferAddress.High,
                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
                    >> 32));
        } else {
                pDiag_release_msg = (pMpi2DiagReleaseRequest_t)
                    (mpt->m_req_frame + (mpt->m_req_frame_size *
                    cmd->cmd_slot));
                bzero(pDiag_release_msg, mpt->m_req_frame_size);
                ddi_put8(mpt->m_acc_req_frame_hdl,
                    &pDiag_release_msg->Function, diag->function);
                ddi_put8(mpt->m_acc_req_frame_hdl,
                    &pDiag_release_msg->BufferType,
                    diag->pBuffer->buffer_type);
        }

        /*
         * Send the message
         */
        (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);
        request_desc_low = (cmd->cmd_slot << 16) +
            MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
        cmd->cmd_rfm = NULL;
        MPTSAS_START_CMD(mpt, request_desc_low, 0);
        if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
            DDI_SUCCESS)) {
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
        }
}

static int
mptsas_post_fw_diag_buffer(mptsas_t *mpt,
    mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code)
{
        mptsas_diag_request_t           diag;
        int                             status, slot_num, post_flags = 0;
        mptsas_cmd_t                    *cmd = NULL;
        struct scsi_pkt                 *pkt;
        pMpi2DiagBufferPostReply_t      reply;
        uint16_t                        iocstatus;
        uint32_t                        iocloginfo, transfer_length;

        /*
         * If buffer is not enabled, just leave.
         */
        *return_code = MPTSAS_FW_DIAG_ERROR_POST_FAILED;
        if (!pBuffer->enabled) {
                status = DDI_FAILURE;
                goto out;
        }

        /*
         * Clear some flags initially.
         */
```

```
        pBuffer->force_release = FALSE;
        pBuffer->valid_data = FALSE;
        pBuffer->owned_by_firmware = FALSE;

        /*
         * Get a cmd buffer from the cmd buffer pool
         */
        if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
                status = DDI_FAILURE;
                mptsas_log(mpt, CE_NOTE, "command pool is full: Post FW Diag");
                goto out;
        }
        post_flags |= MPTSAS_REQUEST_POOL_CMD;

        bzero((caddr_t)cmd, sizeof (*cmd));
        bzero((caddr_t)pkt, scsi_pkt_size());

        cmd->ioc_cmd_slot = (uint32_t)(slot_num);

        diag.pBuffer = pBuffer;
        diag.function = MPI2_FUNCTION_DIAG_BUFFER_POST;

        /*
         * Form a blank cmd/pkt to store the acknowledgement message
         */
        pkt->pkt_ha_private     = (opaque_t)&diag;
        pkt->pkt_flags          = FLAG_HEAD;
        pkt->pkt_time           = 60;
        cmd->cmd_pkt            = pkt;
        cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_DIAG;

        /*
         * Save the command in a slot
         */
        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                /*
                 * Once passthru command get slot, set cmd_flags
                 * CFLAG_PREPARED.
                 */
                cmd->cmd_flags |= CFLAG_PREPARED;
                mptsas_start_diag(mpt, cmd);
        } else {
                mptsas_waitq_add(mpt, cmd);
        }

        while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
                cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
        }

        if (cmd->cmd_flags & CFLAG_TIMEOUT) {
                status = DDI_FAILURE;
                mptsas_log(mpt, CE_WARN, "Post FW Diag command timeout");
                goto out;
        }

        /*
         * cmd_rfm points to the reply message if a reply was given.  Check the
         * IOCStatus to make sure everything went OK with the FW diag request
         * and set buffer flags.
         */
        if (cmd->cmd_rfm) {
                post_flags |= MPTSAS_ADDRESS_REPLY;
                (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
                    DDI_DMA_SYNC_FORCPU);
                reply = (pMpi2DiagBufferPostReply_t)(mpt->m_reply_frame +
                    (cmd->cmd_rfm - mpt->m_reply_frame_dma_addr));
                /*
                 * Get the reply message data
                 */
                iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
                    &reply->IOCStatus);
                iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
                    &reply->IOCLogInfo);
                transfer_length = ddi_get32(mpt->m_acc_reply_frame_hdl,
                    &reply->TransferLength);

                /*
                 * If post failed quit.
                 */
                if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
                        status = DDI_FAILURE;
                        NDBG13(("post FW Diag Buffer failed: IOCStatus=0x%x, "
                            "IOCLogInfo=0x%x, TransferLength=0x%x", iocstatus,
                            iocloginfo, transfer_length));
                        goto out;
                }

                /*
                 * Post was successful.
                 */
                pBuffer->valid_data = TRUE;
                pBuffer->owned_by_firmware = TRUE;
                *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
                status = DDI_SUCCESS;
        }

out:
        /*
         * Put the reply frame back on the free queue, increment the free
         * index, and write the new index to the free index register.  But only
         * if this reply is an ADDRESS reply.
         */
        if (post_flags & MPTSAS_ADDRESS_REPLY) {
                ddi_put32(mpt->m_acc_free_queue_hdl,
                    &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
                    cmd->cmd_rfm);
                (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
                    DDI_DMA_SYNC_FORDEV);
                if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                        mpt->m_free_index = 0;
                }
                ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
                    mpt->m_free_index);
        }
        if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
                mptsas_remove_cmd(mpt, cmd);
                post_flags &= (~MPTSAS_REQUEST_POOL_CMD);
        }
        if (post_flags & MPTSAS_REQUEST_POOL_CMD) {
                mptsas_return_to_pool(mpt, cmd);
        }

        return (status);
}

static int
mptsas_release_fw_diag_buffer(mptsas_t *mpt,
    mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
    uint32_t diag_type)
{
        mptsas_diag_request_t   diag;
```

```
        int                     status, slot_num, rel_flags = 0;
        mptsas_cmd_t            *cmd = NULL;
        struct scsi_pkt         *pkt;
        pMpi2DiagReleaseReply_t reply;
        uint16_t                iocstatus;
        uint32_t                iocloginfo;

        /*
         * If buffer is not enabled, just leave.
         */
        *return_code = MPTSAS_FW_DIAG_ERROR_RELEASE_FAILED;
        if (!pBuffer->enabled) {
                mptsas_log(mpt, CE_NOTE, "This buffer type is not supported "
                    "by the IOC");
                status = DDI_FAILURE;
                goto out;
        }

        /*
         * Clear some flags initially.
         */
        pBuffer->force_release = FALSE;
        pBuffer->valid_data = FALSE;
        pBuffer->owned_by_firmware = FALSE;

        /*
         * Get a cmd buffer from the cmd buffer pool
         */
        if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
                status = DDI_FAILURE;
                mptsas_log(mpt, CE_NOTE, "command pool is full: Release FW "
                    "Diag");
                goto out;
        }
        rel_flags |= MPTSAS_REQUEST_POOL_CMD;

        bzero((caddr_t)cmd, sizeof (*cmd));
        bzero((caddr_t)pkt, scsi_pkt_size());

        cmd->ioc_cmd_slot = (uint32_t)(slot_num);

        diag.pBuffer = pBuffer;
        diag.function = MPI2_FUNCTION_DIAG_RELEASE;

        /*
         * Form a blank cmd/pkt to store the acknowledgement message
         */
        pkt->pkt_ha_private     = (opaque_t)&diag;
        pkt->pkt_flags          = FLAG_HEAD;
        pkt->pkt_time           = 60;
        cmd->cmd_pkt            = pkt;
        cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_DIAG;

        /*
         * Save the command in a slot
         */
        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
                /*
                 * Once passthru command get slot, set cmd_flags
                 * CFLAG_PREPARED.
                 */
                cmd->cmd_flags |= CFLAG_PREPARED;
                mptsas_start_diag(mpt, cmd);
        } else {
                mptsas_waitq_add(mpt, cmd);
        }
        while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
                cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
        }

        if (cmd->cmd_flags & CFLAG_TIMEOUT) {
                status = DDI_FAILURE;
                mptsas_log(mpt, CE_WARN, "Release FW Diag command timeout");
                goto out;
        }

        /*
         * cmd_rfm points to the reply message if a reply was given.  Check the
         * IOCStatus to make sure everything went OK with the FW diag request
         * and set buffer flags.
         */
        if (cmd->cmd_rfm) {
                rel_flags |= MPTSAS_ADDRESS_REPLY;
                (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
                    DDI_DMA_SYNC_FORCPU);
                reply = (pMpi2DiagReleaseReply_t)(mpt->m_reply_frame +
                    (cmd->cmd_rfm - mpt->m_reply_frame_dma_addr));

                /*
                 * Get the reply message data
                 */
                iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
                    &reply->IOCStatus);
                iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
                    &reply->IOCLogInfo);

                /*
                 * If release failed quit.
                 */
                if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) ||
                    pBuffer->owned_by_firmware) {
                        status = DDI_FAILURE;
                        NDBG13(("release FW Diag Buffer failed: "
                            "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
                            iocloginfo));
                        goto out;
                }

                /*
                 * Release was successful.
                 */
                *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
                status = DDI_SUCCESS;

                /*
                 * If this was for an UNREGISTER diag type command, clear the
                 * unique ID.
                 */
                if (diag_type == MPTSAS_FW_DIAG_TYPE_UNREGISTER) {
                        pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
                }
        }

out:
        /*
         * Put the reply frame back on the free queue, increment the free
         * index, and write the new index to the free index register.  But only
         * if this reply is an ADDRESS reply.
         */
        if (rel_flags & MPTSAS_ADDRESS_REPLY) {
                ddi_put32(mpt->m_acc_free_queue_hdl,
```

```
                    &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
                    cmd->cmd_rfm);
                (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
                    DDI_DMA_SYNC_FORDEV);
                if (++mpt->m_free_index == mpt->m_free_queue_depth) {
                        mpt->m_free_index = 0;
                }
                ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
                    mpt->m_free_index);
        }
        if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
                mptsas_remove_cmd(mpt, cmd);
                rel_flags &= (~MPTSAS_REQUEST_POOL_CMD);
        }
        if (rel_flags & MPTSAS_REQUEST_POOL_CMD) {
                mptsas_return_to_pool(mpt, cmd);
        }

        return (status);
}

static int
mptsas_diag_register(mptsas_t *mpt, mptsas_fw_diag_register_t *diag_register,
    uint32_t *return_code)
{
        mptsas_fw_diagnostic_buffer_t   *pBuffer;
        uint8_t                         extended_type, buffer_type, i;
        uint32_t                        buffer_size;
        uint32_t                        unique_id;
        int                             status;

        ASSERT(mutex_owned(&mpt->m_mutex));

        extended_type = diag_register->ExtendedType;
        buffer_type = diag_register->BufferType;
        buffer_size = diag_register->RequestedBufferSize;
        unique_id = diag_register->UniqueId;

        /*
         * Check for valid buffer type
         */
        if (buffer_type >= MPI2_DIAG_BUF_TYPE_COUNT) {
                *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
                return (DDI_FAILURE);
        }

        /*
         * Get the current buffer and look up the unique ID.  The unique ID
         * should not be found.  If it is, the ID is already in use.
         */
        i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
        pBuffer = &mpt->m_fw_diag_buffer_list[buffer_type];
        if (i != MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
                *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
                return (DDI_FAILURE);
        }

        /*
         * The buffer’s unique ID should not be registered yet, and the given
         * unique ID cannot be 0.
         */
        if ((pBuffer->unique_id != MPTSAS_FW_DIAG_INVALID_UID) ||
            (unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
                *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
                return (DDI_FAILURE);
        }
```

```
	/*
	 * If this buffer is already posted as immediate, just change owner.
	 */
	if (pBuffer->immediate && pBuffer->owned_by_firmware &&
	    (pBuffer->unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
		pBuffer->immediate = FALSE;
		pBuffer->unique_id = unique_id;
		return (DDI_SUCCESS);
	}

	/*
	 * Post a new buffer after checking if it's enabled.  The DMA buffer
	 * that is allocated will be contiguous (sgl_len = 1).
	 */
	if (!pBuffer->enabled) {
		*return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
		return (DDI_FAILURE);
	}
	bzero(&pBuffer->buffer_data, sizeof (mptsas_dma_alloc_state_t));
	pBuffer->buffer_data.size = buffer_size;
	if (mptsas_dma_alloc(mpt, &pBuffer->buffer_data) != DDI_SUCCESS) {
		mptsas_log(mpt, CE_WARN, "failed to alloc DMA resource for "
		    "diag buffer: size = %d bytes", buffer_size);
		*return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
		return (DDI_FAILURE);
	}

	/*
	 * Copy the given info to the diag buffer and post the buffer.
	 */
	pBuffer->buffer_type = buffer_type;
	pBuffer->immediate = FALSE;
	if (buffer_type == MPI2_DIAG_BUF_TYPE_TRACE) {
		for (i = 0; i < (sizeof (pBuffer->product_specific) / 4);
		    i++) {
			pBuffer->product_specific[i] =
			    diag_register->ProductSpecific[i];
		}
	}
	pBuffer->extended_type = extended_type;
	pBuffer->unique_id = unique_id;
	status = mptsas_post_fw_diag_buffer(mpt, pBuffer, return_code);

	if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
	    DDI_SUCCESS) {
		mptsas_log(mpt, CE_WARN, "Check of DMA handle failed in "
		    "mptsas_diag_register.");
		ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
			status = DDI_FAILURE;
	}

	/*
	 * In case there was a failure, free the DMA buffer.
	 */
	if (status == DDI_FAILURE) {
		mptsas_dma_free(&pBuffer->buffer_data);
	}

	return (status);
}

static int
mptsas_diag_unregister(mptsas_t *mpt,
    mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code)
{
```

```
	mptsas_fw_diagnostic_buffer_t	*pBuffer;
	uint8_t				i;
	uint32_t			unique_id;
	int				status;

	ASSERT(mutex_owned(&mpt->m_mutex));

	unique_id = diag_unregister->UniqueId;

	/*
	 * Get the current buffer and look up the unique ID.  The unique ID
	 * should be there.
	 */
	i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
	if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
		*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
		return (DDI_FAILURE);
	}

	pBuffer = &mpt->m_fw_diag_buffer_list[i];

	/*
	 * Try to release the buffer from FW before freeing it.  If release
	 * fails, don't free the DMA buffer in case FW tries to access it
	 * later.  If buffer is not owned by firmware, can't release it.
	 */
	if (!pBuffer->owned_by_firmware) {
		status = DDI_SUCCESS;
	} else {
		status = mptsas_release_fw_diag_buffer(mpt, pBuffer,
		    return_code, MPTSAS_FW_DIAG_TYPE_UNREGISTER);
	}

	/*
	 * At this point, return the current status no matter what happens with
	 * the DMA buffer.
	 */
	pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
	if (status == DDI_SUCCESS) {
		if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
		    DDI_SUCCESS) {
			mptsas_log(mpt, CE_WARN, "Check of DMA handle failed "
			    "in mptsas_diag_unregister.");
			ddi_fm_service_impact(mpt->m_dip,
			    DDI_SERVICE_UNAFFECTED);
		}
		mptsas_dma_free(&pBuffer->buffer_data);
	}

	return (status);
}

static int
mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
    uint32_t *return_code)
{
	mptsas_fw_diagnostic_buffer_t	*pBuffer;
	uint8_t				i;
	uint32_t			unique_id;

	ASSERT(mutex_owned(&mpt->m_mutex));

	unique_id = diag_query->UniqueId;

	/*
	 * If ID is valid, query on ID.
	 * If ID is invalid, query on buffer type.
	 */
	if (unique_id == MPTSAS_FW_DIAG_INVALID_UID) {
		i = diag_query->BufferType;
		if (i >= MPI2_DIAG_BUF_TYPE_COUNT) {
			*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
			return (DDI_FAILURE);
		}
	} else {
		i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
		if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
			*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
			return (DDI_FAILURE);
		}
	}

	/*
	 * Fill query structure with the diag buffer info.
	 */
	pBuffer = &mpt->m_fw_diag_buffer_list[i];
	diag_query->BufferType = pBuffer->buffer_type;
	diag_query->ExtendedType = pBuffer->extended_type;
	if (diag_query->BufferType == MPI2_DIAG_BUF_TYPE_TRACE) {
		for (i = 0; i < (sizeof (diag_query->ProductSpecific) / 4);
		    i++) {
			diag_query->ProductSpecific[i] =
			    pBuffer->product_specific[i];
		}
	}
	diag_query->TotalBufferSize = pBuffer->buffer_data.size;
	diag_query->DriverAddedBufferSize = 0;
	diag_query->UniqueId = pBuffer->unique_id;
	diag_query->ApplicationFlags = 0;
	diag_query->DiagnosticFlags = 0;

	/*
	 * Set/Clear application flags
	 */
	if (pBuffer->immediate) {
		diag_query->ApplicationFlags &= ~MPTSAS_FW_DIAG_FLAG_APP_OWNED;
	} else {
		diag_query->ApplicationFlags |= MPTSAS_FW_DIAG_FLAG_APP_OWNED;
	}
	if (pBuffer->valid_data || pBuffer->owned_by_firmware) {
		diag_query->ApplicationFlags |=
		    MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
	} else {
		diag_query->ApplicationFlags &=
		    ~MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
	}
	if (pBuffer->owned_by_firmware) {
		diag_query->ApplicationFlags |=
		    MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
	} else {
		diag_query->ApplicationFlags &=
		    ~MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
	}

	return (DDI_SUCCESS);
}

static int
mptsas_diag_read_buffer(mptsas_t *mpt,
    mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
    uint32_t *return_code, int ioctl_mode)
{
```

```
	mptsas_fw_diagnostic_buffer_t	*pBuffer;
	uint8_t				i, *pData;
	uint32_t			unique_id, byte;
	int				status;

	ASSERT(mutex_owned(&mpt->m_mutex));

	unique_id = diag_read_buffer->UniqueId;

	/*
	 * Get the current buffer and look up the unique ID.  The unique ID
	 * should be there.
	 */
	i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
	if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
		*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
		return (DDI_FAILURE);
	}

	pBuffer = &mpt->m_fw_diag_buffer_list[i];

	/*
	 * Make sure requested read is within limits
	 */
	if (diag_read_buffer->StartingOffset + diag_read_buffer->BytesToRead >
	    pBuffer->buffer_data.size) {
		*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
		return (DDI_FAILURE);
	}

	/*
	 * Copy the requested data from DMA to the diag_read_buffer.  The DMA
	 * buffer that was allocated is one contiguous buffer.
	 */
	pData = (uint8_t *)(pBuffer->buffer_data.memp +
	    diag_read_buffer->StartingOffset);
	(void) ddi_dma_sync(pBuffer->buffer_data.handle, 0, 0,
	    DDI_DMA_SYNC_FORCPU);
	for (byte = 0; byte < diag_read_buffer->BytesToRead; byte++) {
		if (ddi_copyout(pData + byte, ioctl_buf + byte, 1, ioctl_mode)
		    != 0) {
			return (DDI_FAILURE);
		}
	}
	diag_read_buffer->Status = 0;

	/*
	 * Set or clear the Force Release flag.
	 */
	if (pBuffer->force_release) {
		diag_read_buffer->Flags |= MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
	} else {
		diag_read_buffer->Flags &= ~MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
	}

	/*
	 * If buffer is to be reregistered, make sure it's not already owned by
	 * firmware first.
	 */
	status = DDI_SUCCESS;
	if (!pBuffer->owned_by_firmware) {
		if (diag_read_buffer->Flags & MPTSAS_FW_DIAG_FLAG_REREGISTER) {
			status = mptsas_post_fw_diag_buffer(mpt, pBuffer,
			    return_code);
		}
	}
```

```
        return (status);
}

static int
mptsas_diag_release(mptsas_t *mpt, mptsas_fw_diag_release_t *diag_release,
    uint32_t *return_code)
{
        mptsas_fw_diagnostic_buffer_t   *pBuffer;
        uint8_t                         i;
        uint32_t                        unique_id;
        int                             status;

        ASSERT(mutex_owned(&mpt->m_mutex));

        unique_id = diag_release->UniqueId;

        /*
         * Get the current buffer and look up the unique ID.  The unique ID
         * should be there.
         */
        i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
        if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
                *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
                return (DDI_FAILURE);
        }

        pBuffer = &mpt->m_fw_diag_buffer_list[i];

        /*
         * If buffer is not owned by firmware, it’s already been released.
         */
        if (!pBuffer->owned_by_firmware) {
                *return_code = MPTSAS_FW_DIAG_ERROR_ALREADY_RELEASED;
                return (DDI_FAILURE);
        }

        /*
         * Release the buffer.
         */
        status = mptsas_release_fw_diag_buffer(mpt, pBuffer, return_code,
            MPTSAS_FW_DIAG_TYPE_RELEASE);
        return (status);
}

static int
mptsas_do_diag_action(mptsas_t *mpt, uint32_t action, uint8_t *diag_action,
    uint32_t length, uint32_t *return_code, int ioctl_mode)
{
        mptsas_fw_diag_register_t       diag_register;
        mptsas_fw_diag_unregister_t     diag_unregister;
        mptsas_fw_diag_query_t          diag_query;
        mptsas_diag_read_buffer_t       diag_read_buffer;
        mptsas_fw_diag_release_t        diag_release;
        int                             status = DDI_SUCCESS;
        uint32_t                        original_return_code, read_buf_len;

        ASSERT(mutex_owned(&mpt->m_mutex));

        original_return_code = *return_code;
        *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;

        switch (action) {
                case MPTSAS_FW_DIAG_TYPE_REGISTER:
                        if (!length) {
                                *return_code =
```

```
                                    MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
                                status = DDI_FAILURE;
                                break;
                        }
                        if (ddi_copyin(diag_action, &diag_register,
                            sizeof (diag_register), ioctl_mode) != 0) {
                                return (DDI_FAILURE);
                        }
                        status = mptsas_diag_register(mpt, &diag_register,
                            return_code);
                        break;

                case MPTSAS_FW_DIAG_TYPE_UNREGISTER:
                        if (length < sizeof (diag_unregister)) {
                                *return_code =
                                    MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
                                status = DDI_FAILURE;
                                break;
                        }
                        if (ddi_copyin(diag_action, &diag_unregister,
                            sizeof (diag_unregister), ioctl_mode) != 0) {
                                return (DDI_FAILURE);
                        }
                        status = mptsas_diag_unregister(mpt, &diag_unregister,
                            return_code);
                        break;

                case MPTSAS_FW_DIAG_TYPE_QUERY:
                        if (length < sizeof (diag_query)) {
                                *return_code =
                                    MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
                                status = DDI_FAILURE;
                                break;
                        }
                        if (ddi_copyin(diag_action, &diag_query,
                            sizeof (diag_query), ioctl_mode) != 0) {
                                return (DDI_FAILURE);
                        }
                        status = mptsas_diag_query(mpt, &diag_query,
                            return_code);
                        if (status == DDI_SUCCESS) {
                                if (ddi_copyout(&diag_query, diag_action,
                                    sizeof (diag_query), ioctl_mode) != 0) {
                                        return (DDI_FAILURE);
                                }
                        }
                        break;

                case MPTSAS_FW_DIAG_TYPE_READ_BUFFER:
                        if (ddi_copyin(diag_action, &diag_read_buffer,
                            sizeof (diag_read_buffer) - 4, ioctl_mode) != 0) {
                                return (DDI_FAILURE);
                        }
                        read_buf_len = sizeof (diag_read_buffer) -
                            sizeof (diag_read_buffer.DataBuffer) +
                            diag_read_buffer.BytesToRead;
                        if (length < read_buf_len) {
                                *return_code =
                                    MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
                                status = DDI_FAILURE;
                                break;
                        }
                        status = mptsas_diag_read_buffer(mpt,
                            &diag_read_buffer, diag_action +
                            sizeof (diag_read_buffer) - 4, return_code,
                            ioctl_mode);
```

```
			if (status == DDI_SUCCESS) {
				if (ddi_copyout(&diag_read_buffer, diag_action,
				    sizeof (diag_read_buffer) - 4, ioctl_mode)
				    != 0) {
					return (DDI_FAILURE);
				}
			}
			break;

		case MPTSAS_FW_DIAG_TYPE_RELEASE:
			if (length < sizeof (diag_release)) {
				*return_code =
				    MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
				status = DDI_FAILURE;
				break;
			}
			if (ddi_copyin(diag_action, &diag_release,
			    sizeof (diag_release), ioctl_mode) != 0) {
				return (DDI_FAILURE);
			}
			status = mptsas_diag_release(mpt, &diag_release,
			    return_code);
			break;

		default:
			*return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
			status = DDI_FAILURE;
			break;
	}

	if ((status == DDI_FAILURE) &&
	    (original_return_code == MPTSAS_FW_DIAG_NEW) &&
	    (*return_code != MPTSAS_FW_DIAG_ERROR_SUCCESS)) {
		status = DDI_SUCCESS;
	}

	return (status);
}

static int
mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *user_data, int mode)
{
	int			status;
	mptsas_diag_action_t	driver_data;

	ASSERT(mutex_owned(&mpt->m_mutex));

	/*
	 * Copy the user data to a driver data buffer.
	 */
	if (ddi_copyin(user_data, &driver_data, sizeof (mptsas_diag_action_t),
	    mode) == 0) {
		/*
		 * Send diag action request if Action is valid
		 */
		if (driver_data.Action == MPTSAS_FW_DIAG_TYPE_REGISTER ||
		    driver_data.Action == MPTSAS_FW_DIAG_TYPE_UNREGISTER ||
		    driver_data.Action == MPTSAS_FW_DIAG_TYPE_QUERY ||
		    driver_data.Action == MPTSAS_FW_DIAG_TYPE_READ_BUFFER ||
		    driver_data.Action == MPTSAS_FW_DIAG_TYPE_RELEASE) {
			status = mptsas_do_diag_action(mpt, driver_data.Action,
			    (void *)(uintptr_t)driver_data.PtrDiagAction,
			    driver_data.Length, &driver_data.ReturnCode,
			    mode);
			if (status == DDI_SUCCESS) {
				if (ddi_copyout(&driver_data.ReturnCode,
```

```
				    &user_data->ReturnCode,
				    sizeof (user_data->ReturnCode), mode)
				    != 0) {
					status = EFAULT;
				} else {
					status = 0;
				}
			} else {
				status = EIO;
			}
		} else {
			status = EINVAL;
		}
	} else {
		status = EFAULT;
	}

	return (status);
}

/*
 * This routine handles the "event query" ioctl.
 */
static int
mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data, int mode,
    int *rval)
{
	int			status;
	mptsas_event_query_t	driverdata;
	uint8_t			i;

	driverdata.Entries = MPTSAS_EVENT_QUEUE_SIZE;

	mutex_enter(&mpt->m_mutex);
	for (i = 0; i < 4; i++) {
		driverdata.Types[i] = mpt->m_event_mask[i];
	}
	mutex_exit(&mpt->m_mutex);

	if (ddi_copyout(&driverdata, data, sizeof (driverdata), mode) != 0) {
		status = EFAULT;
	} else {
		*rval = MPTIOCTL_STATUS_GOOD;
		status = 0;
	}

	return (status);
}

/*
 * This routine handles the "event enable" ioctl.
 */
static int
mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data, int mode,
    int *rval)
{
	int			status;
	mptsas_event_enable_t	driverdata;
	uint8_t			i;

	if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
		mutex_enter(&mpt->m_mutex);
		for (i = 0; i < 4; i++) {
			mpt->m_event_mask[i] = driverdata.Types[i];
		}
		mutex_exit(&mpt->m_mutex);
```

```
                *rval = MPTIOCTL_STATUS_GOOD;
                status = 0;
        } else {
                status = EFAULT;
        }
        return (status);
}

/*
 * This routine handles the "event report" ioctl.
 */
static int
mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data, int mode,
    int *rval)
{
        int                     status;
        mptsas_event_report_t   driverdata;

        mutex_enter(&mpt->m_mutex);

        if (ddi_copyin(&data->Size, &driverdata.Size, sizeof (driverdata.Size),
            mode) == 0) {
                if (driverdata.Size >= sizeof (mpt->m_events)) {
                        if (ddi_copyout(mpt->m_events, data->Events,
                            sizeof (mpt->m_events), mode) != 0) {
                                status = EFAULT;
                        } else {
                                if (driverdata.Size > sizeof (mpt->m_events)) {
                                        driverdata.Size =
                                            sizeof (mpt->m_events);
                                        if (ddi_copyout(&driverdata.Size,
                                            &data->Size,
                                            sizeof (driverdata.Size),
                                            mode) != 0) {
                                                status = EFAULT;
                                        } else {
                                                *rval = MPTIOCTL_STATUS_GOOD;
                                                status = 0;
                                        }
                                } else {
                                        *rval = MPTIOCTL_STATUS_GOOD;
                                        status = 0;
                                }
                        }
                } else {
                        *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
                        status = 0;
                }
        } else {
                status = EFAULT;
        }

        mutex_exit(&mpt->m_mutex);
        return (status);
}

static void
mptsas_lookup_pci_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
{
        int     *reg_data;
        uint_t  reglen;

        /*
         * Lookup the 'reg' property and extract the other data
         */
```

```
        if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
            DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
            DDI_PROP_SUCCESS) {
                /*
                 * Extract the PCI data from the 'reg' property first DWORD.
                 * The entry looks like the following:
                 * First DWORD:
                 * Bits 0 - 7 8-bit Register number
                 * Bits 8 - 10 3-bit Function number
                 * Bits 11 - 15 5-bit Device number
                 * Bits 16 - 23 8-bit Bus number
                 * Bits 24 - 25 2-bit Address Space type identifier
                 *
                 */
                adapter_data->PciInformation.u.bits.BusNumber =
                    (reg_data[0] & 0x00FF0000) >> 16;
                adapter_data->PciInformation.u.bits.DeviceNumber =
                    (reg_data[0] & 0x0000F800) >> 11;
                adapter_data->PciInformation.u.bits.FunctionNumber =
                    (reg_data[0] & 0x00000700) >> 8;
                ddi_prop_free((void *)reg_data);
        } else {
                /*
                 * If we can't determine the PCI data then we fill in FF's for
                 * the data to indicate this.
                 */
                adapter_data->PCIDeviceHwId = 0xFFFFFFFF;
                adapter_data->MpiPortNumber = 0xFFFFFFFF;
                adapter_data->PciInformation.u.AsDWORD = 0xFFFFFFFF;
        }

        /*
         * Saved in the mpt->m_fwversion
         */
        adapter_data->MpiFirmwareVersion = mpt->m_fwversion;
}

static void
mptsas_read_adapter_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
{
        char    *driver_verstr = MPTSAS_MOD_STRING;

        mptsas_lookup_pci_data(mpt, adapter_data);
        adapter_data->AdapterType = mpt->m_MPI25 ?
            MPTIOCTL_ADAPTER_TYPE_SAS3 :
            MPTIOCTL_ADAPTER_TYPE_SAS2;
        adapter_data->PCIDeviceHwId = (uint32_t)mpt->m_devid;
        adapter_data->PCIDeviceHwRev = (uint32_t)mpt->m_revid;
        adapter_data->SubSystemId = (uint32_t)mpt->m_ssid;
        adapter_data->SubsystemVendorId = (uint32_t)mpt->m_svid;
        (void) strcpy((char *)&adapter_data->DriverVersion[0], driver_verstr);
        adapter_data->BiosVersion = 0;
        (void) mptsas_get_bios_page3(mpt, &adapter_data->BiosVersion);
}

static void
mptsas_read_pci_info(mptsas_t *mpt, mptsas_pci_info_t *pci_info)
{
        int     *reg_data, i;
        uint_t  reglen;

        /*
         * Lookup the 'reg' property and extract the other data
         */
        if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
            DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
	    DDI_PROP_SUCCESS) {
		/*
		 * Extract the PCI data from the ’reg’ property first DWORD.
		 * The entry looks like the following:
		 * First DWORD:
		 * Bits 8 - 10 3-bit Function number
		 * Bits 11 - 15 5-bit Device number
		 * Bits 16 - 23 8-bit Bus number
		 */
		pci_info->BusNumber = (reg_data[0] & 0x00FF0000) >> 16;
		pci_info->DeviceNumber = (reg_data[0] & 0x0000F800) >> 11;
		pci_info->FunctionNumber = (reg_data[0] & 0x00000700) >> 8;
		ddi_prop_free((void *)reg_data);
	} else {
		/*
		 * If we can’t determine the PCI info then we fill in FF’s for
		 * the data to indicate this.
		 */
		pci_info->BusNumber = 0xFFFFFFFF;
		pci_info->DeviceNumber = 0xFF;
		pci_info->FunctionNumber = 0xFF;
	}

	/*
	 * Now get the interrupt vector and the pci header.  The vector can
	 * only be 0 right now.  The header is the first 256 bytes of config
	 * space.
	 */
	pci_info->InterruptVector = 0;
	for (i = 0; i < sizeof (pci_info->PciHeader); i++) {
		pci_info->PciHeader[i] = pci_config_get8(mpt->m_config_handle,
		    i);
	}
}

static int
mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data, int mode)
{
	int			status = 0;
	mptsas_reg_access_t	driverdata;

	mutex_enter(&mpt->m_mutex);
	if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
		switch (driverdata.Command) {
			/*
			 * IO access is not supported.
			 */
			case REG_IO_READ:
			case REG_IO_WRITE:
				mptsas_log(mpt, CE_WARN, "IO access is not "
				    "supported.  Use memory access.");
				status = EINVAL;
				break;

			case REG_MEM_READ:
				driverdata.RegData = ddi_get32(mpt->m_datap,
				    (uint32_t *)(void *)mpt->m_reg +
				    driverdata.RegOffset);
				if (ddi_copyout(&driverdata.RegData,
				    &data->RegData,
				    sizeof (driverdata.RegData), mode) != 0) {
					mptsas_log(mpt, CE_WARN, "Register "
					    "Read Failed");
					status = EFAULT;
				}
				break;
```

```

			case REG_MEM_WRITE:
				ddi_put32(mpt->m_datap,
				    (uint32_t *)(void *)mpt->m_reg +
				    driverdata.RegOffset,
				    driverdata.RegData);
				break;

			default:
				status = EINVAL;
				break;
		}
	} else {
		status = EFAULT;
	}

	mutex_exit(&mpt->m_mutex);
	return (status);
}

static int
led_control(mptsas_t *mpt, intptr_t data, int mode)
{
	int ret = 0;
	mptsas_led_control_t lc;
	mptsas_target_t *ptgt;

	if (ddi_copyin((void *)data, &lc, sizeof (lc), mode) != 0) {
		return (EFAULT);
	}

	if ((lc.Command != MPTSAS_LEDCTL_FLAG_SET &&
	    lc.Command != MPTSAS_LEDCTL_FLAG_GET) ||
	    lc.Led < MPTSAS_LEDCTL_LED_MIN ||
	    lc.Led > MPTSAS_LEDCTL_LED_MAX ||
	    (lc.Command == MPTSAS_LEDCTL_FLAG_SET && lc.LedStatus != 0 &&
	    lc.LedStatus != 1)) {
		return (EINVAL);
	}

	if ((lc.Command == MPTSAS_LEDCTL_FLAG_SET && (mode & FWRITE) == 0) ||
	    (lc.Command == MPTSAS_LEDCTL_FLAG_GET && (mode & FREAD) == 0))
		return (EACCES);

	/* Locate the target we’re interrogating... */
	mutex_enter(&mpt->m_mutex);
	ptgt = refhash_linear_search(mpt->m_targets,
	    mptsas_target_eval_slot, &lc);
	if (ptgt == NULL) {
		/* We could not find a target for that enclosure/slot. */
		mutex_exit(&mpt->m_mutex);
		return (ENOENT);
	}

	if (lc.Command == MPTSAS_LEDCTL_FLAG_SET) {
		/* Update our internal LED state. */
		ptgt->m_led_status &= ~(1 << (lc.Led - 1));
		ptgt->m_led_status |= lc.LedStatus << (lc.Led - 1);

		/* Flush it to the controller. */
		ret = mptsas_flush_led_status(mpt, ptgt);
		mutex_exit(&mpt->m_mutex);
		return (ret);
	}

	/* Return our internal LED state. */
```

```
        lc.LedStatus = (ptgt->m_led_status >> (lc.Led - 1)) & 1;
        mutex_exit(&mpt->m_mutex);

        if (ddi_copyout(&lc, (void *)data, sizeof (lc), mode) != 0) {
                return (EFAULT);
        }

        return (0);
}

static int
get_disk_info(mptsas_t *mpt, intptr_t data, int mode)
{
        uint16_t i = 0;
        uint16_t count = 0;
        int ret = 0;
        mptsas_target_t *ptgt;
        mptsas_disk_info_t *di;
        STRUCT_DECL(mptsas_get_disk_info, gdi);

        if ((mode & FREAD) == 0)
                return (EACCES);

        STRUCT_INIT(gdi, get_udatamodel());

        if (ddi_copyin((void *)data, STRUCT_BUF(gdi), STRUCT_SIZE(gdi),
            mode) != 0) {
                return (EFAULT);
        }

        /* Find out how many targets there are. */
        mutex_enter(&mpt->m_mutex);
        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                count++;
        }
        mutex_exit(&mpt->m_mutex);

        /*
         * If we haven't been asked to copy out information on each target,
         * then just return the count.
         */
        STRUCT_FSET(gdi, DiskCount, count);
        if (STRUCT_FGETP(gdi, PtrDiskInfoArray) == NULL)
                goto copy_out;

        /*
         * If we haven't been given a large enough buffer to copy out into,
         * let the caller know.
         */
        if (STRUCT_FGET(gdi, DiskInfoArraySize) <
            count * sizeof (mptsas_disk_info_t)) {
                ret = ENOSPC;
                goto copy_out;
        }

        di = kmem_zalloc(count * sizeof (mptsas_disk_info_t), KM_SLEEP);

        mutex_enter(&mpt->m_mutex);
        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                if (i >= count) {
                        /*
                         * The number of targets changed while we weren't
                         * looking, so give up.
                         */
```

```
                        refhash_rele(mpt->m_targets, ptgt);
                        mutex_exit(&mpt->m_mutex);
                        kmem_free(di, count * sizeof (mptsas_disk_info_t));
                        return (EAGAIN);
                }
                di[i].Instance = mpt->m_instance;
                di[i].Enclosure = ptgt->m_enclosure;
                di[i].Slot = ptgt->m_slot_num;
                di[i].SasAddress = ptgt->m_addr.mta_wwn;
                i++;
        }
        mutex_exit(&mpt->m_mutex);
        STRUCT_FSET(gdi, DiskCount, i);

        /* Copy out the disk information to the caller. */
        if (ddi_copyout((void *)di, STRUCT_FGETP(gdi, PtrDiskInfoArray),
            i * sizeof (mptsas_disk_info_t), mode) != 0) {
                ret = EFAULT;
        }

        kmem_free(di, count * sizeof (mptsas_disk_info_t));

copy_out:
        if (ddi_copyout(STRUCT_BUF(gdi), (void *)data, STRUCT_SIZE(gdi),
            mode) != 0) {
                ret = EFAULT;
        }

        return (ret);
}

static int
mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
    int *rval)
{
        int                     status = 0;
        mptsas_t                *mpt;
        mptsas_update_flash_t   flashdata;
        mptsas_pass_thru_t      passthru_data;
        mptsas_adapter_data_t   adapter_data;
        mptsas_pci_info_t       pci_info;
        int                     copylen;

        int                     iport_flag = 0;
        dev_info_t              *dip = NULL;
        mptsas_phymask_t        phymask = 0;
        struct devctl_iocdata   *dcp = NULL;
        char                    *addr = NULL;
        mptsas_target_t         *ptgt = NULL;

        *rval = MPTIOCTL_STATUS_GOOD;
        if (secpolicy_sys_config(credp, B_FALSE) != 0) {
                return (EPERM);
        }

        mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
        if (mpt == NULL) {
                /*
                 * Called from iport node, get the states
                 */
                iport_flag = 1;
                dip = mptsas_get_dip_from_dev(dev, &phymask);
                if (dip == NULL) {
                        return (ENXIO);
                }
                mpt = DIP2MPT(dip);
```

```
	}
	/* Make sure power level is D0 before accessing registers */
	mutex_enter(&mpt->m_mutex);
	if (mpt->m_options & MPTSAS_OPT_PM) {
		(void) pm_busy_component(mpt->m_dip, 0);
		if (mpt->m_power_level != PM_LEVEL_D0) {
			mutex_exit(&mpt->m_mutex);
			if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
			    DDI_SUCCESS) {
				mptsas_log(mpt, CE_WARN,
				    "mptsas%d: mptsas_ioctl: Raise power "
				    "request failed.", mpt->m_instance);
				(void) pm_idle_component(mpt->m_dip, 0);
				return (ENXIO);
			}
		} else {
			mutex_exit(&mpt->m_mutex);
		}
	} else {
		mutex_exit(&mpt->m_mutex);
	}

	if (iport_flag) {
		status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
		if (status != 0) {
			goto out;
		}
		/*
		 * The following code control the OK2RM LED, it doesn’t affect
		 * the ioctl return status.
		 */
		if ((cmd == DEVCTL_DEVICE_ONLINE) ||
		    (cmd == DEVCTL_DEVICE_OFFLINE)) {
			if (ndi_dc_allochdl((void *)data, &dcp) !=
			    NDI_SUCCESS) {
				goto out;
			}
			addr = ndi_dc_getaddr(dcp);
			ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
			if (ptgt == NULL) {
				NDBG14(("mptsas_ioctl led control: tgt %s not "
				    "found", addr));
				ndi_dc_freehdl(dcp);
				goto out;
			}
			mutex_enter(&mpt->m_mutex);
			if (cmd == DEVCTL_DEVICE_ONLINE) {
				ptgt->m_tgt_unconfigured = 0;
			} else if (cmd == DEVCTL_DEVICE_OFFLINE) {
				ptgt->m_tgt_unconfigured = 1;
			}
			if (cmd == DEVCTL_DEVICE_OFFLINE) {
				ptgt->m_led_status |=
				    (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
			} else {
				ptgt->m_led_status &=
				    ~(1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
			}
			(void) mptsas_flush_led_status(mpt, ptgt);
			mutex_exit(&mpt->m_mutex);
			ndi_dc_freehdl(dcp);
		}
		goto out;
	}
	switch (cmd) {
		case MPTIOCTL_GET_DISK_INFO:
```

```
			status = get_disk_info(mpt, data, mode);
			break;
		case MPTIOCTL_LED_CONTROL:
			status = led_control(mpt, data, mode);
			break;
		case MPTIOCTL_UPDATE_FLASH:
			if (ddi_copyin((void *)data, &flashdata,
				sizeof (struct mptsas_update_flash), mode)) {
				status = EFAULT;
				break;
			}

			mutex_enter(&mpt->m_mutex);
			if (mptsas_update_flash(mpt,
			    (caddr_t)(long)flashdata.PtrBuffer,
			    flashdata.ImageSize, flashdata.ImageType, mode)) {
				status = EFAULT;
			}

			/*
			 * Reset the chip to start using the new
			 * firmware.  Reset if failed also.
			 */
			mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
			if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
				status = EFAULT;
			}
			mutex_exit(&mpt->m_mutex);
			break;
		case MPTIOCTL_PASS_THRU:
			/*
			 * The user has requested to pass through a command to
			 * be executed by the MPT firmware.  Call our routine
			 * which does this.  Only allow one passthru IOCTL at
			 * one time. Other threads will block on
			 * m_passthru_mutex, which is of adaptive variant.
			 */
			if (ddi_copyin((void *)data, &passthru_data,
			    sizeof (mptsas_pass_thru_t), mode)) {
				status = EFAULT;
				break;
			}
			mutex_enter(&mpt->m_passthru_mutex);
			mutex_enter(&mpt->m_mutex);
			status = mptsas_pass_thru(mpt, &passthru_data, mode);
			mutex_exit(&mpt->m_mutex);
			mutex_exit(&mpt->m_passthru_mutex);

			break;
		case MPTIOCTL_GET_ADAPTER_DATA:
			/*
			 * The user has requested to read adapter data.  Call
			 * our routine which does this.
			 */
			bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
			if (ddi_copyin((void *)data, (void *)&adapter_data,
			    sizeof (mptsas_adapter_data_t), mode)) {
				status = EFAULT;
				break;
			}
			if (adapter_data.StructureLength >=
			    sizeof (mptsas_adapter_data_t)) {
				adapter_data.StructureLength = (uint32_t)
				    sizeof (mptsas_adapter_data_t);
				copylen = sizeof (mptsas_adapter_data_t);
				mutex_enter(&mpt->m_mutex);
				mptsas_read_adapter_data(mpt, &adapter_data);
				mutex_exit(&mpt->m_mutex);
			} else {
				adapter_data.StructureLength = (uint32_t)
				    sizeof (mptsas_adapter_data_t);
				copylen = sizeof (adapter_data.StructureLength);
				*rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
			}
			if (ddi_copyout((void *)(&adapter_data), (void *)data,
			    copylen, mode) != 0) {
				status = EFAULT;
			}
			break;
		case MPTIOCTL_GET_PCI_INFO:
			/*
			 * The user has requested to read pci info.  Call
			 * our routine which does this.
			 */
			bzero(&pci_info, sizeof (mptsas_pci_info_t));
			mutex_enter(&mpt->m_mutex);
			mptsas_read_pci_info(mpt, &pci_info);
			mutex_exit(&mpt->m_mutex);
			if (ddi_copyout((void *)(&pci_info), (void *)data,
			    sizeof (mptsas_pci_info_t), mode) != 0) {
				status = EFAULT;
			}
			break;
		case MPTIOCTL_RESET_ADAPTER:
			mutex_enter(&mpt->m_mutex);
			mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
			if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
				mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
				    "failed");
				status = EFAULT;
			}
			mutex_exit(&mpt->m_mutex);
			break;
		case MPTIOCTL_DIAG_ACTION:
			/*
			 * The user has done a diag buffer action.  Call our
			 * routine which does this.  Only allow one diag action
			 * at one time.
			 */
			mutex_enter(&mpt->m_mutex);
			if (mpt->m_diag_action_in_progress) {
				mutex_exit(&mpt->m_mutex);
				return (EBUSY);
			}
			mpt->m_diag_action_in_progress = 1;
			status = mptsas_diag_action(mpt,
			    (mptsas_diag_action_t *)data, mode);
			mpt->m_diag_action_in_progress = 0;
			mutex_exit(&mpt->m_mutex);
			break;
		case MPTIOCTL_EVENT_QUERY:
			/*
			 * The user has done an event query. Call our routine
			 * which does this.
			 */
			status = mptsas_event_query(mpt,
			    (mptsas_event_query_t *)data, mode, rval);
			break;
		case MPTIOCTL_EVENT_ENABLE:
			/*
			 * The user has done an event enable. Call our routine
			 * which does this.
```

```
			 */
			status = mptsas_event_enable(mpt,
			    (mptsas_event_enable_t *)data, mode, rval);
			break;
		case MPTIOCTL_EVENT_REPORT:
			/*
			 * The user has done an event report. Call our routine
			 * which does this.
			 */
			status = mptsas_event_report(mpt,
			    (mptsas_event_report_t *)data, mode, rval);
			break;
		case MPTIOCTL_REG_ACCESS:
			/*
			 * The user has requested register access.  Call our
			 * routine which does this.
			 */
			status = mptsas_reg_access(mpt,
			    (mptsas_reg_access_t *)data, mode);
			break;
		default:
			status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
			    rval);
			break;
	}

out:
	return (status);
}

int
mptsas_restart_ioc(mptsas_t *mpt)
{
	int		rval = DDI_SUCCESS;
	mptsas_target_t *ptgt = NULL;

	ASSERT(mutex_owned(&mpt->m_mutex));

	/*
	 * Set a flag telling I/O path that we're processing a reset.  This is
	 * needed because after the reset is complete, the hash table still
	 * needs to be rebuilt.  If I/Os are started before the hash table is
	 * rebuilt, I/O errors will occur.  This flag allows I/Os to be marked
	 * so that they can be retried.
	 */
	mpt->m_in_reset = TRUE;

	/*
	 * Set all throttles to HOLD
	 */
	for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
	    ptgt = refhash_next(mpt->m_targets, ptgt)) {
		mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
	}

	/*
	 * Disable interrupts
	 */
	MPTSAS_DISABLE_INTR(mpt);

	/*
	 * Abort all commands: outstanding commands, commands in waitq and
	 * tx_waitq.
	 */
	mptsas_flush_hba(mpt);
```

```
        /*
         * Reinitialize the chip.
         */
        if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
                rval = DDI_FAILURE;
        }

        /*
         * Enable interrupts again
         */
        MPTSAS_ENABLE_INTR(mpt);

        /*
         * If mptsas_init_chip was successful, update the driver data.
         */
        if (rval == DDI_SUCCESS) {
                mptsas_update_driver_data(mpt);
        }

        /*
         * Reset the throttles
         */
        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
            ptgt = refhash_next(mpt->m_targets, ptgt)) {
                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
        }

        mptsas_doneq_empty(mpt);
        mptsas_restart_hba(mpt);

        if (rval != DDI_SUCCESS) {
                mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
        }

        /*
         * Clear the reset flag so that I/Os can continue.
         */
        mpt->m_in_reset = FALSE;

        return (rval);
}

static int
mptsas_init_chip(mptsas_t *mpt, int first_time)
{
        ddi_dma_cookie_t        cookie;
        uint32_t                i;
        int                     rval;

        /*
         * Check to see if the firmware image is valid
         */
        if (ddi_get32(mpt->m_datap, &mpt->m_reg->HostDiagnostic) &
            MPI2_DIAG_FLASH_BAD_SIG) {
                mptsas_log(mpt, CE_WARN, "mptsas bad flash signature!");
                goto fail;
        }

        /*
         * Reset the chip
         */
        rval = mptsas_ioc_reset(mpt, first_time);
        if (rval == MPTSAS_RESET_FAIL) {
                mptsas_log(mpt, CE_WARN, "hard reset failed!");
                goto fail;
```

```
        }

        if ((rval == MPTSAS_SUCCESS_MUR) && (!first_time)) {
                goto mur;
        }
        /*
         * Setup configuration space
         */
        if (mptsas_config_space_init(mpt) == FALSE) {
                mptsas_log(mpt, CE_WARN, "mptsas_config_space_init "
                    "failed!");
                goto fail;
        }

        /*
         * IOC facts can change after a diag reset so all buffers that are
         * based on these numbers must be de-allocated and re-allocated.  Get
         * new IOC facts each time chip is initialized.
         */
        if (mptsas_ioc_get_facts(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_ioc_get_facts failed");
                goto fail;
        }

       mpt->m_targets = refhash_create(MPTSAS_TARGET_BUCKET_COUNT,
           mptsas_target_addr_hash, mptsas_target_addr_cmp,
           mptsas_target_free, sizeof (mptsas_target_t),
           offsetof(mptsas_target_t, m_link),
           offsetof(mptsas_target_t, m_addr), KM_SLEEP);

        if (mptsas_alloc_active_slots(mpt, KM_SLEEP)) {
                goto fail;
        }
        /*
         * Allocate request message frames, reply free queue, reply descriptor
         * post queue, and reply message frames using latest IOC facts.
         */
        if (mptsas_alloc_request_frames(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_alloc_request_frames failed");
                goto fail;
        }
        if (mptsas_alloc_free_queue(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_alloc_free_queue failed!");
                goto fail;
        }
        if (mptsas_alloc_post_queue(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_alloc_post_queue failed!");
                goto fail;
        }
        if (mptsas_alloc_reply_frames(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_alloc_reply_frames failed!");
                goto fail;
        }

mur:
        /*
         * Re-Initialize ioc to operational state
         */
        if (mptsas_ioc_init(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_ioc_init failed");
                goto fail;
        }

        mptsas_alloc_reply_args(mpt);

        /*
```

```
         * Initialize reply post index.  Reply free index is initialized after
         * the next loop.
         */
        mpt->m_post_index = 0;

        /*
         * Initialize the Reply Free Queue with the physical addresses of our
         * reply frames.
         */
        cookie.dmac_address = mpt->m_reply_frame_dma_addr;
        for (i = 0; i < mpt->m_max_replies; i++) {
                ddi_put32(mpt->m_acc_free_queue_hdl,
                    &((uint32_t *)(void *)mpt->m_free_queue)[i],
                    cookie.dmac_address);
                cookie.dmac_address += mpt->m_reply_frame_size;
        }
        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);

        /*
         * Initialize the reply free index to one past the last frame on the
         * queue.  This will signify that the queue is empty to start with.
         */
        mpt->m_free_index = i;
        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex, i);

        /*
         * Initialize the reply post queue to 0xFFFFFFFF,0xFFFFFFFF's.
         */
        for (i = 0; i < mpt->m_post_queue_depth; i++) {
                ddi_put64(mpt->m_acc_post_queue_hdl,
                    &((uint64_t *)(void *)mpt->m_post_queue)[i],
                    0xFFFFFFFFFFFFFFFF);
        }
        (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
            DDI_DMA_SYNC_FORDEV);

        /*
         * Enable ports
         */
        if (mptsas_ioc_enable_port(mpt) == DDI_FAILURE) {
                mptsas_log(mpt, CE_WARN, "mptsas_ioc_enable_port failed");
                goto fail;
        }

        /*
         * enable events
         */
        if (mptsas_ioc_enable_event_notification(mpt)) {
                goto fail;
        }

        /*
         * We need checks in attach and these.
         * chip_init is called in mult. places
         */

        if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=
```

```
            DDI_SUCCESS)) {
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
                goto fail;
        }

        /* Check all acc handles */
        if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
            DDI_SUCCESS) ||
            (mptsas_check_acc_handle(mpt->m_config_handle) !=
            DDI_SUCCESS)) {
                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
                goto fail;
        }

        return (DDI_SUCCESS);

fail:
        return (DDI_FAILURE);
}
______unchanged_portion_omitted_

void
mptsas_update_driver_data(struct mptsas *mpt)
{
        mptsas_target_t *tp;
        mptsas_smp_t *sp;

        ASSERT(MUTEX_HELD(&mpt->m_mutex));

        /*
         * TODO after hard reset, update the driver data structures
         * 1. update port/phymask mapping table mpt->m_phy_info
         * 2. invalid all the entries in hash table
         *    m_devhdl = 0xffff and m_deviceinfo = 0
         * 3. call sas_device_page/expander_page to update hash table
         */
        mptsas_update_phymask(mpt);

#endif /* ! codereview */
        /*
         * Remove all the devhdls for existing entries but leave their
         * addresses alone.  In update_hashtab() below, we'll find all
         * targets that are still present and reassociate them with
         * their potentially new devhdls.  Leaving the targets around in
         * this fashion allows them to be used on the tx waitq even
         * while IOC reset it occurring.
        * Invalid the existing entries
        *
        * XXX - It seems like we should just delete everything here.  We are
        * holding the lock and are about to refresh all the targets in both
        * hashes anyway.  Given the path we're in, what outstanding async
        * event could possibly be trying to reference one of these things
        * without taking the lock, and how would that be useful anyway?
         */
        for (tp = refhash_first(mpt->m_targets); tp != NULL;
            tp = refhash_next(mpt->m_targets, tp)) {
                tp->m_devhdl = MPTSAS_INVALID_DEVHDL;
                tp->m_deviceinfo = 0;
                tp->m_dr_flag = MPTSAS_DR_INACTIVE;
        }
        for (sp = refhash_first(mpt->m_smp_targets); sp != NULL;
            sp = refhash_next(mpt->m_smp_targets, sp)) {
                sp->m_devhdl = MPTSAS_INVALID_DEVHDL;
                sp->m_deviceinfo = 0;
        }
        mpt->m_done_traverse_dev = 0;
        mpt->m_done_traverse_smp = 0;
        mpt->m_dev_handle = mpt->m_smp_devhdl = MPTSAS_INVALID_DEVHDL;
        mptsas_update_hashtab(mpt);
}
```
______***unchanged_portion_omitted_***