```
**********************************************************
  414477 Tue Dec  4 16:30:32 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
re #10443 rb3479 3.1.3 crash: BAD TRAP: type=e (#pf Page fault)
**********************************************************
______unchanged_portion_omitted_

2623 static void
2624 mptsas_alloc_reply_args(mptsas_t *mpt)
2625 {
2626         if (mpt->m_replyh_args == NULL) {
2626         if (mpt->m_replyh_args != NULL) {
2627                 kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
2628                     * mpt->m_max_replies);
2629                 mpt->m_replyh_args = NULL;
2630         }
2627                 mpt->m_replyh_args = kmem_zalloc(sizeof (m_replyh_arg_t) *
2628                     mpt->m_max_replies, KM_SLEEP);
2629         }
2630 }
______unchanged_portion_omitted_

4704 static void
4705 mptsas_handle_address_reply(mptsas_t *mpt,
4706     pMpi2ReplyDescriptorsUnion_t reply_desc)
4707 {
4708         pMpi2AddressReplyDescriptor_t   address_reply;
4709         pMPI2DefaultReply_t             reply;
4710         mptsas_fw_diagnostic_buffer_t   *pBuffer;
4711         uint32_t                        reply_addr;
4712         uint16_t                        SMID, iocstatus;
4713         mptsas_slots_t                  *slots = mpt->m_active;
4714         mptsas_cmd_t                    *cmd = NULL;
4715         uint8_t                         function, buffer_type;
4716         m_replyh_arg_t                  *args;
4717         int                             reply_frame_no;

4719         ASSERT(mutex_owned(&mpt->m_mutex));

4721         address_reply = (pMpi2AddressReplyDescriptor_t)reply_desc;
4722         reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
4723             &address_reply->ReplyFrameAddress);
4724         SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

4726         /*
4727          * If reply frame is not in the proper range we should ignore this
4728          * message and exit the interrupt handler.
4729          */
4730         if ((reply_addr < mpt->m_reply_frame_dma_addr) ||
4731             (reply_addr >= (mpt->m_reply_frame_dma_addr +
4732             (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
4733             ((reply_addr - mpt->m_reply_frame_dma_addr) %
4734             mpt->m_reply_frame_size != 0)) {
4735                 mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
4736                     "address 0x%x\n", reply_addr);
4737                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4738                 return;
4739         }

4741         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
4742             DDI_DMA_SYNC_FORCPU);
4743         reply = (pMPI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
4744             mpt->m_reply_frame_dma_addr));
4745         function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

4747         /*
```



```
4748          * don't get slot information and command for events since these values
4749          * don't exist
4750          */
4751         if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
4752             (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
4753                 /*
4754                  * This could be a TM reply, which use the last allocated SMID,
4755                  * so allow for that.
4756                  */
4757                 if ((SMID == 0) || (SMID > (slots->m_n_slots + 1))) {
4758                         mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
4759                             "%d\n", SMID);
4760                         ddi_fm_service_impact(mpt->m_dip,
4761                             DDI_SERVICE_UNAFFECTED);
4762                         return;
4763                 }

4765                 cmd = slots->m_slot[SMID];

4767                 /*
4768                  * print warning and return if the slot is empty
4769                  */
4770                 if (cmd == NULL) {
4771                         mptsas_log(mpt, CE_WARN, "?NULL command for address "
4772                             "reply in slot %d", SMID);
4773                         return;
4774                 }
4775                 if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
4776                     (cmd->cmd_flags & CFLAG_CONFIG) ||
4777                     (cmd->cmd_flags & CFLAG_FW_DIAG)) {
4778                         cmd->cmd_rfm = reply_addr;
4779                         cmd->cmd_flags |= CFLAG_FINISHED;
4780                         cv_broadcast(&mpt->m_passthru_cv);
4781                         cv_broadcast(&mpt->m_config_cv);
4782                         cv_broadcast(&mpt->m_fw_diag_cv);
4783                         return;
4784                 } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
4785                         mptsas_remove_cmd(mpt, cmd);
4786                 }
4787                 NDBG31(("\t\tmptsas_process_intr: slot=%d", SMID));
4788         }
4789         /*
4790          * Depending on the function, we need to handle
4791          * the reply frame (and cmd) differently.
4792          */
4793         switch (function) {
4794         case MPI2_FUNCTION_SCSI_IO_REQUEST:
4795                 mptsas_check_scsi_io_error(mpt, (pMpi2SCSIIOReply_t)reply, cmd);
4796                 break;
4797         case MPI2_FUNCTION_SCSI_TASK_MGMT:
4798                 cmd->cmd_rfm = reply_addr;
4799                 mptsas_check_task_mgt(mpt, (pMpi2SCSIManagementReply_t)reply,
4800                     cmd);
4801                 break;
4802         case MPI2_FUNCTION_FW_DOWNLOAD:
4803                 cmd->cmd_flags |= CFLAG_FINISHED;
4804                 cv_signal(&mpt->m_fw_cv);
4805                 break;
4806         case MPI2_FUNCTION_EVENT_NOTIFICATION:
4807                 reply_frame_no = (reply_addr - mpt->m_reply_frame_dma_addr) /
4808                     mpt->m_reply_frame_size;
4809                 args = &mpt->m_replyh_args[reply_frame_no];
4810                 args->mpt = (void *)mpt;
4811                 args->rfm = reply_addr;

4813                 /*
```

```
4814                 * Record the event if its type is enabled in
4815                 * this mpt instance by ioctl.
4816                 */
4817                mptsas_record_event(args);

4819                /*
4820                 * Handle time critical events
4821                 * NOT_RESPONDING/ADDED only now
4822                 */
4823                if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {
4824                        /*
4825                         * Would not return main process,
4826                         * just let taskq resolve ack action
4827                         * and ack would be sent in taskq thread
4828                         */
4829                        NDBG20(("send mptsas_handle_event_sync success"));
4830                }

4832                if (mpt->m_in_reset) {
4833                        NDBG20(("dropping event received during reset"));
4834                        return;
4835                }

4837                if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
4838                    (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
4839                        mptsas_log(mpt, CE_WARN, "No memory available"
4840                        "for dispatch taskq");
4841                        /*
4842                         * Return the reply frame to the free queue.
4843                         */
4844                        ddi_put32(mpt->m_acc_free_queue_hdl,
4845                            &((uint32_t *)(void *)
4846                            mpt->m_free_queue)[mpt->m_free_index], reply_addr);
4847                        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4848                            DDI_DMA_SYNC_FORDEV);
4849                        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4850                                mpt->m_free_index = 0;
4851                        }

4853                        ddi_put32(mpt->m_datap,
4854                            &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
4855                }
4856                return;
4857        case MPI2_FUNCTION_DIAG_BUFFER_POST:
4858                /*
4859                 * If SMID is 0, this implies that the reply is due to a
4860                 * release function with a status that the buffer has been
4861                 * released.  Set the buffer flags accordingly.
4862                 */
4863                if (SMID == 0) {
4864                        iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
4865                            &reply->IOCStatus);
4866                        buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
4867                            &(((pMpi2DiagBufferPostReply_t)reply)->BufferType));
4868                        if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
4869                                pBuffer =
4870                                    &mpt->m_fw_diag_buffer_list[buffer_type];
4871                                pBuffer->valid_data = TRUE;
4872                                pBuffer->owned_by_firmware = FALSE;
4873                                pBuffer->immediate = FALSE;
4874                        }
4875                } else {
4876                        /*
4877                         * Normal handling of diag post reply with SMID.
4878                         */
4879                        cmd = slots->m_slot[SMID];
```



```
4881                        /*
4882                         * print warning and return if the slot is empty
4883                         */
4884                        if (cmd == NULL) {
4885                                mptsas_log(mpt, CE_WARN, "?NULL command for "
4886                                    "address reply in slot %d", SMID);
4887                                return;
4888                        }
4889                        cmd->cmd_rfm = reply_addr;
4890                        cmd->cmd_flags |= CFLAG_FINISHED;
4891                        cv_broadcast(&mpt->m_fw_diag_cv);
4892                }
4893                return;
4894        default:
4895                mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
4896                break;
4897        }

4899        /*
4900         * Return the reply frame to the free queue.
4901         */
4902        ddi_put32(mpt->m_acc_free_queue_hdl,
4903            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
4904            reply_addr);
4905        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4906            DDI_DMA_SYNC_FORDEV);
4907        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4908                mpt->m_free_index = 0;
4909        }
4910        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
4911            mpt->m_free_index);

4913        if (cmd->cmd_flags & CFLAG_FW_CMD)
4914                return;

4916        if (cmd->cmd_flags & CFLAG_RETRY) {
4917                /*
4918                 * The target returned QFULL or busy, do not add tihs
4919                 * pkt to the doneq since the hba will retry
4920                 * this cmd.
4921                 *
4922                 * The pkt has already been resubmitted in
4923                 * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4924                 * Remove this cmd_flag here.
4925                 */
4926                cmd->cmd_flags &= ~CFLAG_RETRY;
4927        } else {
4928                mptsas_doneq_add(mpt, cmd);
4929        }
4930 }
```

*_____unchanged_portion_omitted_*

```
5608 /*
5609  * mptsas_handle_dr is a task handler for DR, the DR action includes:
5610  * 1. Directly attched Device Added/Removed.
5611  * 2. Expander Device Added/Removed.
5612  * 3. Indirectly Attached Device Added/Expander.
5613  * 4. LUNs of a existing device status change.
5614  * 5. RAID volume created/deleted.
5615  * 6. Member of RAID volume is released because of RAID deletion.
5616  * 7. Physical disks are removed because of RAID creation.
5617  */
5618 static void
5619 mptsas_handle_dr(void *args) {
5620         mptsas_topo_change_list_t       *topo_node = NULL;
```

```
5621        mptsas_topo_change_list_t       *save_node = NULL;
5622        mptsas_t                        *mpt;
5623        dev_info_t                      *parent = NULL;
5624        mptsas_phymask_t                phymask = 0;
5625        char                            *phy_mask_name;
5626        uint8_t                         flags = 0, physport = 0xff;
5627        uint8_t                         port_update = 0;
5628        uint_t                          event;

5630        topo_node = (mptsas_topo_change_list_t *)args;

5632        mpt = topo_node->mpt;
5633        event = topo_node->event;
5634        flags = topo_node->flags;

5636        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);

5638        NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));

5640        switch (event) {
5641        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5642                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5643                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
5644                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
5645                        /*
5646                         * Direct attached or expander attached device added
5647                         * into system or a Phys Disk that is being unhidden.
5648                         */
5649                        port_update = 1;
5650                }
5651                break;
5652        case MPTSAS_DR_EVENT_RECONFIG_SMP:
5653                /*
5654                 * New expander added into system, it must be the head
5655                 * of topo_change_list_t
5656                 */
5657                port_update = 1;
5658                break;
5659        default:
5660                port_update = 0;
5661                break;
5662        }
5663        /*
5664         * All cases port_update == 1 may cause initiator port form change
5665         */
5666        mutex_enter(&mpt->m_mutex);
5667        if (mpt->m_port_chng && port_update) {
5668                /*
5669                 * mpt->m_port_chng flag indicates some PHYs of initiator
5670                 * port have changed to online. So when expander added or
5671                 * directly attached device online event come, we force to
5672                 * update port information by issueing SAS IO Unit Page and
5673                 * update PHYMASKs.
5674                 */
5675                (void) mptsas_update_phymask(mpt);
5676                mpt->m_port_chng = 0;

5678        }
5679        mutex_exit(&mpt->m_mutex);
5680        while (topo_node) {
5681                phymask = 0;
5682                if (parent == NULL) {
5683                        physport = topo_node->un.physport;
5684                        event = topo_node->event;
5685                        flags = topo_node->flags;
5686                        if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
```



```
5687                            MPTSAS_DR_EVENT_OFFLINE_SMP)) {
5688                                /*
5689                                 * For all offline events, phymask is known
5690                                 */
5691                                phymask = topo_node->un.phymask;
5692                                goto find_parent;
5693                        }
5694                        if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
5695                                goto handle_topo_change;
5696                        }
5697                        if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
5698                                phymask = topo_node->un.phymask;
5699                                goto find_parent;
5700                        }

5702                        if ((flags ==
5703                            MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
5704                            (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
5705                                /*
5706                                 * There is no any field in IR_CONFIG_CHANGE
5707                                 * event indicate physport/phynum, let's get
5708                                 * parent after SAS Device Page0 request.
5709                                 */
5710                                goto handle_topo_change;
5711                        }

5713                        mutex_enter(&mpt->m_mutex);
5714                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5715                                /*
5716                                 * If the direct attached device added or a
5717                                 * phys disk is being unhidden, argument
5718                                 * physport actually is PHY#, so we have to get
5719                                 * phymask according PHY#.
5720                                 */
5721                                physport = mpt->m_phy_info[physport].port_num;
5722                        }

5724                        /*
5725                         * Translate physport to phymask so that we can search
5726                         * parent dip.
5727                         */
5728                        phymask = mptsas_physport_to_phymask(mpt,
5729                            physport);
5730                        mutex_exit(&mpt->m_mutex);

5732 find_parent:
5733                        bzero(phy_mask_name, MPTSAS_MAX_PHYS);
5734                        /*
5735                         * For RAID topology change node, write the iport name
5736                         * as v0.
5737                         */
5738                        if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5739                                (void) sprintf(phy_mask_name, "v0");
5740                        } else {
5741                                /*
5742                                 * phymask can bo 0 if the drive has been
5743                                 * pulled by the time an add event is
5744                                 * processed.  If phymask is 0, just skip this
5745                                 * event and continue.
5746                                 */
5747                                if (phymask == 0) {
5748                                        mutex_enter(&mpt->m_mutex);
5749                                        save_node = topo_node;
5750                                        topo_node = topo_node->next;
5751                                        ASSERT(save_node);
5752                                        kmem_free(save_node,
```

```
5753                                            sizeof (mptsas_topo_change_list_t));
5754                                    mutex_exit(&mpt->m_mutex);
5755
5756                                    parent = NULL;
5757                                    continue;
5758                            }
5759                            (void) sprintf(phy_mask_name, "%x", phymask);
5760                    }
5761                    parent = scsi_hba_iport_find(mpt->m_dip,
5762                        phy_mask_name);
5763                    if (parent == NULL) {
5764                            mptsas_log(mpt, CE_WARN, "Failed to find an "
5765                                "iport, should not happen!");
5766                            goto out;
5767                    }
5768
5769                }
5770                ASSERT(parent);
5771 handle_topo_change:
5772
5773                mutex_enter(&mpt->m_mutex);
5774                /*
5775                 * If HBA is being reset, don't perform operations depending
5776                 * on the IOC. We must free the topo list, however.
5777                 */
5778                if (!mpt->m_in_reset)
5779                        mptsas_handle_topo_change(topo_node, parent);
5780                else
5781                        NDBG20(("skipping topo change received during reset"));
5782                save_node = topo_node;
5783                topo_node = topo_node->next;
5784                ASSERT(save_node);
5785                kmem_free(save_node, sizeof (mptsas_topo_change_list_t));
5786                mutex_exit(&mpt->m_mutex);
5787
5788                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5789                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
5790                    (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) {
5791                        /*
5792                         * If direct attached device associated, make sure
5793                         * reset the parent before start the next one. But
5794                         * all devices associated with expander shares the
5795                         * parent.  Also, reset parent if this is for RAID.
5796                         */
5797                        parent = NULL;
5798                }
5799        }
5800 out:
5801        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5802 }
```

*unchanged_portion_omitted_*

```
6984 /*
6985  * handle events from ioc
6986  */
6987 static void
6988 mptsas_handle_event(void *args)
6989 {
6990        m_replyh_arg_t                  *replyh_arg;
6991        pMpi2EventNotificationReply_t   eventreply;
6992        uint32_t                        event, iocloginfo, rfm;
6993        uint32_t                        status;
6994        uint8_t                         port;
6995        mptsas_t                        *mpt;
6996        uint_t                          iocstatus;
```



```
6998        replyh_arg = (m_replyh_arg_t *)args;
6999        rfm = replyh_arg->rfm;
7000        mpt = replyh_arg->mpt;
7001
7002        mutex_enter(&mpt->m_mutex);
7003        /*
7004         * If HBA is being reset, drop incoming event.
7005         */
7006        if (mpt->m_in_reset) {
7007                NDBG20(("dropping event received prior to reset"));
7008                mutex_exit(&mpt->m_mutex);
7009                return;
7010        }
7011
7012        eventreply = (pMpi2EventNotificationReply_t)
7013            (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
7014        event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);
7015
7016        if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
7017            &eventreply->IOCStatus)) {
7018                if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
7019                        mptsas_log(mpt, CE_WARN,
7020                            "!mptsas_handle_event: IOCStatus=0x%x, "
7021                            "IOCLogInfo=0x%x", iocstatus,
7022                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7023                            &eventreply->IOCLogInfo));
7024                } else {
7025                        mptsas_log(mpt, CE_WARN,
7026                            "mptsas_handle_event: IOCStatus=0x%x, "
7027                            "IOCLogInfo=0x%x", iocstatus,
7028                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7029                            &eventreply->IOCLogInfo));
7030                }
7031        }
7032
7033        /*
7034         * figure out what kind of event we got and handle accordingly
7035         */
7036        switch (event) {
7037        case MPI2_EVENT_LOG_ENTRY_ADDED:
7038                break;
7039        case MPI2_EVENT_LOG_DATA:
7040                iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7041                    &eventreply->IOCLogInfo);
7042                NDBG20(("mptsas %d log info %x received.\n", mpt->m_instance,
7043                    iocloginfo));
7044                break;
7045        case MPI2_EVENT_STATE_CHANGE:
7046                NDBG20(("mptsas%d state change.", mpt->m_instance));
7047                break;
7048        case MPI2_EVENT_HARD_RESET_RECEIVED:
7049                NDBG20(("mptsas%d event change.", mpt->m_instance));
7050                break;
7051        case MPI2_EVENT_SAS_DISCOVERY:
7052        {
7053                MPI2_EVENT_DATA_SAS_DISCOVERY   *sasdiscovery;
7054                char                            string[80];
7055                uint8_t                         rc;
7056
7057                sasdiscovery =
7058                    (pMpi2EventDataSasDiscovery_t)eventreply->EventData;
7059
7060                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7061                    &sasdiscovery->ReasonCode);
7062                port = ddi_get8(mpt->m_acc_reply_frame_hdl,
```

```
7063                    &sasdiscovery->PhysicalPort);
7064                status = ddi_get32(mpt->m_acc_reply_frame_hdl,
7065                    &sasdiscovery->DiscoveryStatus);

7067                string[0] = 0;
7068                switch (rc) {
7069                case MPI2_EVENT_SAS_DISC_RC_STARTED:
7070                        (void) sprintf(string, "STARTING");
7071                        break;
7072                case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
7073                        (void) sprintf(string, "COMPLETED");
7074                        break;
7075                default:
7076                        (void) sprintf(string, "UNKNOWN");
7077                        break;
7078                }

7080                NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
7081                    port, status));

7083                break;
7084        }
7085        case MPI2_EVENT_EVENT_CHANGE:
7086                NDBG20(("mptsas%d event change.", mpt->m_instance));
7087                break;
7088        case MPI2_EVENT_TASK_SET_FULL:
7089        {
7090                pMpi2EventDataTaskSetFull_t     taskfull;

7092                taskfull = (pMpi2EventDataTaskSetFull_t)eventreply->EventData;

7094                NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
7095                    mpt->m_instance,  ddi_get16(mpt->m_acc_reply_frame_hdl,
7096                    &taskfull->CurrentDepth)));
7097                break;
7098        }
7099        case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
7100        {
7101                /*
7102                 * SAS TOPOLOGY CHANGE LIST Event has already been handled
7103                 * in mptsas_handle_event_sync() of interrupt context
7104                 */
7105                break;
7106        }
7107        case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
7108        {
7109                pMpi2EventDataSasEnclDevStatusChange_t  encstatus;
7110                uint8_t                                 rc;
7111                char                                    string[80];

7113                encstatus = (pMpi2EventDataSasEnclDevStatusChange_t)
7114                    eventreply->EventData;

7116                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7117                    &encstatus->ReasonCode);
7118                switch (rc) {
7119                case MPI2_EVENT_SAS_ENCL_RC_ADDED:
7120                        (void) sprintf(string, "added");
7121                        break;
7122                case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
7123                        (void) sprintf(string, ", not responding");
7124                        break;
7125                default:
7126                break;
7127                }
7128                NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure %x%s\n",
```



```
7129                    mpt->m_instance, ddi_get16(mpt->m_acc_reply_frame_hdl,
7130                    &encstatus->EnclosureHandle), string));
7131                break;
7132        }

7134        /*
7135         * MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by
7136         * mptsas_handle_event_sync,in here just send ack message.
7137         */
7138        case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
7139        {
7140                pMpi2EventDataSasDeviceStatusChange_t   statuschange;
7141                uint8_t                                 rc;
7142                uint16_t                                devhdl;
7143                uint64_t                                wwn = 0;
7144                uint32_t                                wwn_lo, wwn_hi;

7146                statuschange = (pMpi2EventDataSasDeviceStatusChange_t)
7147                    eventreply->EventData;
7148                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7149                    &statuschange->ReasonCode);
7150                wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7151                    (uint32_t *)(void *)&statuschange->SASAddress);
7152                wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
7153                    (uint32_t *)(void *)&statuschange->SASAddress + 1);
7154                wwn = ((uint64_t)wwn_hi << 32) | wwn_lo;
7155                devhdl =  ddi_get16(mpt->m_acc_reply_frame_hdl,
7156                    &statuschange->DevHandle);

7158                NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wwn is %"PRIx64,
7159                    wwn));

7161                switch (rc) {
7162                case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
7163                        NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
7164                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7165                            &statuschange->ASC),
7166                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7167                            &statuschange->ASCQ)));
7168                        break;

7170                case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
7171                        NDBG20(("Device not supported"));
7172                        break;

7174                case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
7175                        NDBG20(("IOC internally generated the Target Reset "
7176                            "for devhdl:%x", devhdl));
7177                        break;

7179                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
7180                        NDBG20(("IOC's internally generated Target Reset "
7181                            "completed for devhdl:%x", devhdl));
7182                        break;

7184                case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
7185                        NDBG20(("IOC internally generated Abort Task"));
7186                        break;

7188                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
7189                        NDBG20(("IOC's internally generated Abort Task "
7190                            "completed"));
7191                        break;

7193                case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
7194                        NDBG20(("IOC internally generated Abort Task Set"));
```

```
7195                        break;
7196
7197                case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
7198                        NDBG20(("IOC internally generated Clear Task Set"));
7199                        break;
7200
7201                case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
7202                        NDBG20(("IOC internally generated Query Task"));
7203                        break;
7204
7205                case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
7206                        NDBG20(("Device sent an Asynchronous Notification"));
7207                        break;
7208
7209                default:
7210                        break;
7211                }
7212                break;
7213        }
7214        case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7215        {
7216                /*
7217                 * IR TOPOLOGY CHANGE LIST Event has already been handled
7218                 * in mpt_handle_event_sync() of interrupt context
7219                 */
7220                break;
7221        }
7222        case MPI2_EVENT_IR_OPERATION_STATUS:
7223        {
7224                Mpi2EventDataIrOperationStatus_t        *irOpStatus;
7225                char                                    reason_str[80];
7226                uint8_t                                 rc, percent;
7227                uint16_t                                handle;
7228
7229                irOpStatus = (pMpi2EventDataIrOperationStatus_t)
7230                    eventreply->EventData;
7231                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7232                    &irOpStatus->RAIDOperation);
7233                percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
7234                    &irOpStatus->PercentComplete);
7235                handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7236                    &irOpStatus->VolDevHandle);
7237
7238                switch (rc) {
7239                        case MPI2_EVENT_IR_RAIDOP_RESYNC:
7240                                (void) sprintf(reason_str, "resync");
7241                                break;
7242                        case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
7243                                (void) sprintf(reason_str, "online capacity "
7244                                    "expansion");
7245                                break;
7246                        case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
7247                                (void) sprintf(reason_str, "consistency check");
7248                                break;
7249                        default:
7250                                (void) sprintf(reason_str, "unknown reason %x",
7251                                    rc);
7252                }
7253
7254                NDBG20(("mptsas%d raid operational status: (%s)"
7255                    "\thandle(0x%04x), percent complete(%d)\n",
7256                    mpt->m_instance, reason_str, handle, percent));
7257                break;
7258        }
7259        case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
7260        {
```



```
7261                pMpi2EventDataSasBroadcastPrimitive_t   sas_broadcast;
7262                uint8_t                                 phy_num;
7263                uint8_t                                 primitive;
7264
7265                sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
7266                    eventreply->EventData;
7267
7268                phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
7269                    &sas_broadcast->PhyNum);
7270                primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
7271                    &sas_broadcast->Primitive);
7272
7273                switch (primitive) {
7274                case MPI2_EVENT_PRIMITIVE_CHANGE:
7275                        mptsas_smhba_log_sysevent(mpt,
7276                            ESC_SAS_HBA_PORT_BROADCAST,
7277                            SAS_PORT_BROADCAST_CHANGE,
7278                            &mpt->m_phy_info[phy_num].smhba_info);
7279                        break;
7280                case MPI2_EVENT_PRIMITIVE_SES:
7281                        mptsas_smhba_log_sysevent(mpt,
7282                            ESC_SAS_HBA_PORT_BROADCAST,
7283                            SAS_PORT_BROADCAST_SES,
7284                            &mpt->m_phy_info[phy_num].smhba_info);
7285                        break;
7286                case MPI2_EVENT_PRIMITIVE_EXPANDER:
7287                        mptsas_smhba_log_sysevent(mpt,
7288                            ESC_SAS_HBA_PORT_BROADCAST,
7289                            SAS_PORT_BROADCAST_D01_4,
7290                            &mpt->m_phy_info[phy_num].smhba_info);
7291                        break;
7292                case MPI2_EVENT_PRIMITIVE_ASYNCHRONOUS_EVENT:
7293                        mptsas_smhba_log_sysevent(mpt,
7294                            ESC_SAS_HBA_PORT_BROADCAST,
7295                            SAS_PORT_BROADCAST_D04_7,
7296                            &mpt->m_phy_info[phy_num].smhba_info);
7297                        break;
7298                case MPI2_EVENT_PRIMITIVE_RESERVED3:
7299                        mptsas_smhba_log_sysevent(mpt,
7300                            ESC_SAS_HBA_PORT_BROADCAST,
7301                            SAS_PORT_BROADCAST_D16_7,
7302                            &mpt->m_phy_info[phy_num].smhba_info);
7303                        break;
7304                case MPI2_EVENT_PRIMITIVE_RESERVED4:
7305                        mptsas_smhba_log_sysevent(mpt,
7306                            ESC_SAS_HBA_PORT_BROADCAST,
7307                            SAS_PORT_BROADCAST_D29_7,
7308                            &mpt->m_phy_info[phy_num].smhba_info);
7309                        break;
7310                case MPI2_EVENT_PRIMITIVE_CHANGE0_RESERVED:
7311                        mptsas_smhba_log_sysevent(mpt,
7312                            ESC_SAS_HBA_PORT_BROADCAST,
7313                            SAS_PORT_BROADCAST_D24_0,
7314                            &mpt->m_phy_info[phy_num].smhba_info);
7315                        break;
7316                case MPI2_EVENT_PRIMITIVE_CHANGE1_RESERVED:
7317                        mptsas_smhba_log_sysevent(mpt,
7318                            ESC_SAS_HBA_PORT_BROADCAST,
7319                            SAS_PORT_BROADCAST_D27_4,
7320                            &mpt->m_phy_info[phy_num].smhba_info);
7321                        break;
7322                default:
7323                        NDBG20(("mptsas%d: unknown BROADCAST PRIMITIVE"
7324                            " %x received",
7325                            mpt->m_instance, primitive));
7326                        break;
```

```
7327                }
7328                NDBG20(("mptsas%d sas broadcast primitive: "
7329                    "\tprimitive(0x%04x), phy(%d) complete\n",
7330                    mpt->m_instance, primitive, phy_num));
7331                break;
7332        }
7333        case MPI2_EVENT_IR_VOLUME:
7334        {
7335                Mpi2EventDataIrVolume_t         *irVolume;
7336                uint16_t                        devhandle;
7337                uint32_t                        state;
7338                int                             config, vol;
7339                mptsas_slots_t                  *slots = mpt->m_active;
7340                uint8_t                         found = FALSE;

7342                irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
7343                state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7344                    &irVolume->NewValue);
7345                devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7346                    &irVolume->VolDevHandle);

7348                NDBG20(("EVENT_IR_VOLUME event is received"));

7350                /*
7351                 * Get latest RAID info and then find the DevHandle for this
7352                 * event in the configuration.  If the DevHandle is not found
7353                 * just exit the event.
7354                 */
7355                (void) mptsas_get_raid_info(mpt);
7356                for (config = 0; (config < slots->m_num_raid_configs) &&
7357                    (!found); config++) {
7358                        for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
7359                                if (slots->m_raidconfig[config].m_raidvol[vol].
7360                                    m_raidhandle == devhandle) {
7361                                        found = TRUE;
7362                                        break;
7363                                }
7364                        }
7365                }
7366                if (!found) {
7367                        break;
7368                }

7370                switch (irVolume->ReasonCode) {
7371                case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
7372                {
7373                        uint32_t i;
7374                        slots->m_raidconfig[config].m_raidvol[vol].m_settings =
7375                            state;

7377                        i = state & MPI2_RAIDVOL0_SETTING_MASK_WRITE_CACHING;
7378                        mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
7379                            ", auto-config of hot-swap drives is %s"
7380                            ", write caching is %s"
7381                            ", hot-spare pool mask is %02x\n",
7382                            vol, state &
7383                            MPI2_RAIDVOL0_SETTING_AUTO_CONFIG_HSWAP_DISABLE
7384                            ? "disabled" : "enabled",
7385                            i == MPI2_RAIDVOL0_SETTING_UNCHANGED
7386                            ? "controlled by member disks" :
7387                            i == MPI2_RAIDVOL0_SETTING_DISABLE_WRITE_CACHING
7388                            ? "disabled" :
7389                            i == MPI2_RAIDVOL0_SETTING_ENABLE_WRITE_CACHING
7390                            ? "enabled" :
7391                            "incorrectly set",
7392                            (state >> 16) & 0xff);
```



```
7393                                break;
7394                }
7395                case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
7396                {
7397                        slots->m_raidconfig[config].m_raidvol[vol].m_state =
7398                            (uint8_t)state;

7400                        mptsas_log(mpt, CE_NOTE,
7401                            "Volume %d is now %s\n", vol,
7402                            state == MPI2_RAID_VOL_STATE_OPTIMAL
7403                            ? "optimal" :
7404                            state == MPI2_RAID_VOL_STATE_DEGRADED
7405                            ? "degraded" :
7406                            state == MPI2_RAID_VOL_STATE_ONLINE
7407                            ? "online" :
7408                            state == MPI2_RAID_VOL_STATE_INITIALIZING
7409                            ? "initializing" :
7410                            state == MPI2_RAID_VOL_STATE_FAILED
7411                            ? "failed" :
7412                            state == MPI2_RAID_VOL_STATE_MISSING
7413                            ? "missing" :
7414                            "state unknown");
7415                        break;
7416                }
7417                case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
7418                {
7419                        slots->m_raidconfig[config].m_raidvol[vol].
7420                            m_statusflags = state;

7422                        mptsas_log(mpt, CE_NOTE,
7423                            " Volume %d is now %s%s%s%s%s%s%s%s\n",
7424                            vol,
7425                            state & MPI2_RAIDVOL0_STATUS_FLAG_ENABLED
7426                            ? ", enabled" : ", disabled",
7427                            state & MPI2_RAIDVOL0_STATUS_FLAG_QUIESCED
7428                            ? ", quiesced" : "",
7429                            state & MPI2_RAIDVOL0_STATUS_FLAG_VOLUME_INACTIVE
7430                            ? ", inactive" : ", active",
7431                            state &
7432                            MPI2_RAIDVOL0_STATUS_FLAG_BAD_BLOCK_TABLE_FULL
7433                            ? ", bad block table is full" : "",
7434                            state &
7435                            MPI2_RAIDVOL0_STATUS_FLAG_RESYNC_IN_PROGRESS
7436                            ? ", resync in progress" : "",
7437                            state & MPI2_RAIDVOL0_STATUS_FLAG_BACKGROUND_INIT
7438                            ? ", background initialization in progress" : "",
7439                            state &
7440                            MPI2_RAIDVOL0_STATUS_FLAG_CAPACITY_EXPANSION
7441                            ? ", capacity expansion in progress" : "",
7442                            state &
7443                            MPI2_RAIDVOL0_STATUS_FLAG_CONSISTENCY_CHECK
7444                            ? ", consistency check in progress" : "",
7445                            state & MPI2_RAIDVOL0_STATUS_FLAG_DATA_SCRUB
7446                            ? ", data scrub in progress" : "");
7447                        break;
7448                }
7449                default:
7450                        break;
7451                }
7452                break;
7453        }
7454        case MPI2_EVENT_IR_PHYSICAL_DISK:
7455        {
7456                Mpi2EventDataIrPhysicalDisk_t   *irPhysDisk;
7457                uint16_t                        devhandle, enchandle, slot;
7458                uint32_t                        status, state;
```

```
7459                uint8_t                         physdisknum, reason;

7461                irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
7462                    eventreply->EventData;
7463                physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,
7464                    &irPhysDisk->PhysDiskNum);
7465                devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7466                    &irPhysDisk->PhysDiskDevHandle);
7467                enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7468                    &irPhysDisk->EnclosureHandle);
7469                slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
7470                    &irPhysDisk->Slot);
7471                state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7472                    &irPhysDisk->NewValue);
7473                reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
7474                    &irPhysDisk->ReasonCode);

7476                NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));

7478                switch (reason) {
7479                case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
7480                        mptsas_log(mpt, CE_NOTE,
7481                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7482                            "for enclosure with handle 0x%x is now in hot "
7483                            "spare pool %d",
7484                            physdisknum, devhandle, slot, enchandle,
7485                            (state >> 16) & 0xff);
7486                        break;

7488                case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
7489                        status = state;
7490                        mptsas_log(mpt, CE_NOTE,
7491                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7492                            "for enclosure with handle 0x%x is now "
7493                            "%s%s%s%s%s\n", physdisknum, devhandle, slot,
7494                            enchandle,
7495                            status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
7496                            ? ", inactive" : ", active",
7497                            status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
7498                            ? ", out of sync" : "",
7499                            status & MPI2_PHYSDISK0_STATUS_FLAG_QUIESCED
7500                            ? ", quiesced" : "",
7501                            status &
7502                            MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
7503                            ? ", write cache enabled" : "",
7504                            status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
7505                            ? ", capacity expansion target" : "");
7506                        break;

7508                case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
7509                        mptsas_log(mpt, CE_NOTE,
7510                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7511                            "for enclosure with handle 0x%x is now %s\n",
7512                            physdisknum, devhandle, slot, enchandle,
7513                            state == MPI2_RAID_PD_STATE_OPTIMAL
7514                            ? "optimal" :
7515                            state == MPI2_RAID_PD_STATE_REBUILDING
7516                            ? "rebuilding" :
7517                            state == MPI2_RAID_PD_STATE_DEGRADED
7518                            ? "degraded" :
7519                            state == MPI2_RAID_PD_STATE_HOT_SPARE
7520                            ? "a hot spare" :
7521                            state == MPI2_RAID_PD_STATE_ONLINE
7522                            ? "online" :
7523                            state == MPI2_RAID_PD_STATE_OFFLINE
7524                            ? "offline" :
```



```
7525                            state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
7526                            ? "not compatible" :
7527                            state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
7528                            ? "not configured" :
7529                            "state unknown");
7530                        break;
7531                }
7532                break;
7533        }
7534        default:
7535                NDBG20(("mptsas%d: unknown event %x received",
7536                    mpt->m_instance, event));
7537                break;
7538        }

7540        /*
7541         * Return the reply frame to the free queue.
7542         */
7543        ddi_put32(mpt->m_acc_free_queue_hdl,
7544            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index], rfm);
7545        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
7546            DDI_DMA_SYNC_FORDEV);
7547        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
7548                mpt->m_free_index = 0;
7549        }
7550        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
7551            mpt->m_free_index);
7552        mutex_exit(&mpt->m_mutex);
7553 }
```

_____***unchanged_portion_omitted_***

```
8653 /*
8654  * Clean up hba state, abort all outstanding command and commands in waitq
8655  * reset timeout of all targets.
8656  */
8657 static void
8658 mptsas_flush_hba(mptsas_t *mpt)
8659 {
8660        mptsas_slots_t  *slots = mpt->m_active;
8661        mptsas_cmd_t    *cmd;
8662        int             slot;

8664        NDBG25(("mptsas_flush_hba"));

8666        /*
8667         * The I/O Controller should have already sent back
8668         * all commands via the scsi I/O reply frame.  Make
8669         * sure all commands have been flushed.
8670         * Account for TM request, which use the last SMID.
8671         */
8672        for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8673                if ((cmd = slots->m_slot[slot]) == NULL)
8674                        continue;

8676                if (cmd->cmd_flags & CFLAG_CMDIOC) {
8677                        /*
8678                         * Need to make sure to tell everyone that might be
8679                         * waiting on this command that it's going to fail.  If
8680                         * we get here, this command will never timeout because
8681                         * the active command table is going to be re-allocated,
8682                         * so there will be nothing to check against a time out.
8683                         * Instead, mark the command as failed due to reset.
8684                         */
8685                        mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
8686                            STAT_BUS_RESET);
8687                        if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
```

```
8688                            (cmd->cmd_flags & CFLAG_CONFIG) ||
8689                            (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8690                                cmd->cmd_flags |= CFLAG_FINISHED;
8691                                cv_broadcast(&mpt->m_passthru_cv);
8692                                cv_broadcast(&mpt->m_config_cv);
8693                                cv_broadcast(&mpt->m_fw_diag_cv);
8694                        }
8695                        continue;
8696                }

8698                NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
8699                    slot));
8700                mptsas_dump_cmd(mpt, cmd);

8702                mptsas_remove_cmd(mpt, cmd);
8703                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8704                mptsas_doneq_add(mpt, cmd);
8705        }

8707        /*
8708         * Flush the waitq.
8709         */
8710        while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
8711                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8712                if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8713                    (cmd->cmd_flags & CFLAG_CONFIG) ||
8714                    (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8715                        cmd->cmd_flags |= CFLAG_FINISHED;
8716                        cv_broadcast(&mpt->m_passthru_cv);
8717                        cv_broadcast(&mpt->m_config_cv);
8718                        cv_broadcast(&mpt->m_fw_diag_cv);
8719                } else {
8720                        mptsas_doneq_add(mpt, cmd);
8721                }
8722        }

8724        /*
8725         * Flush the tx_waitq
8726         */
8727        mutex_enter(&mpt->m_tx_waitq_mutex);
8728        while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
8729                mutex_exit(&mpt->m_tx_waitq_mutex);
8730                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8731                mptsas_doneq_add(mpt, cmd);
8732                mutex_enter(&mpt->m_tx_waitq_mutex);
8733        }
8734        mutex_exit(&mpt->m_tx_waitq_mutex);

8736        /*
8737         * Drain the taskqs prior to reallocating resources.
8738         */
8739        mutex_exit(&mpt->m_mutex);
8740        ddi_taskq_wait(mpt->m_event_taskq);
8741        ddi_taskq_wait(mpt->m_dr_taskq);
8742        mutex_enter(&mpt->m_mutex);
8743 }
```

*_____unchanged_portion_omitted_*