```
**********************************************************
  449709 Mon Dec 22 09:51:44 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
First attempt at pulling 4310 fix from Andy Giles tree
**********************************************************
______unchanged_portion_omitted_
9810 #endif

9812 /*
9813  * timeout handling
9814  */
9815 static void
9816 mptsas_watch(void *arg)
9817 {
9818 #ifndef __lock_lint
9819         _NOTE(ARGUNUSED(arg))
9820 #endif

9822         mptsas_t        *mpt;
9823         uint32_t        doorbell;

9825         NDBG30(("mptsas_watch"));

9827         rw_enter(&mptsas_global_rwlock, RW_READER);
9828         for (mpt = mptsas_head; mpt != (mptsas_t *)NULL; mpt = mpt->m_next) {

9830                 mutex_enter(&mpt->m_mutex);

9832                 /* Skip device if not powered on */
9833                 if (mpt->m_options & MPTSAS_OPT_PM) {
9834                         if (mpt->m_power_level == PM_LEVEL_D0) {
9835                                 (void) pm_busy_component(mpt->m_dip, 0);
9836                                 mpt->m_busy = 1;
9837                         } else {
9838                                 mutex_exit(&mpt->m_mutex);
9839                                 continue;
9840                         }
9841                 }

9843                 /*
9844                  * Check if controller is in a FAULT state. If so, reset it.
9845                  */
9846                 doorbell = ddi_get32(mpt->m_datap, &mpt->m_reg->Doorbell);
9847                 if ((doorbell & MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_FAULT) {
9848                         doorbell &= MPI2_DOORBELL_DATA_MASK;
9849                         mptsas_log(mpt, CE_WARN, "MPT Firmware Fault, "
9850                             "code: %04x", doorbell);
9851                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
9852                         if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
9853                                 mptsas_log(mpt, CE_WARN, "Reset failed"
9854                                     "after fault was detected");
9855                         }
9856                 }
9857
9858                 /*
9859                  * If we set the "please reset me" flag, then reset.
9860                  */
9861                 if (mpt->m_softstate & MPTSAS_SS_RESET_INWATCH) {
9862                         doorbell = ddi_get32(mpt->m_datap,
9863                             &mpt->m_reg->Doorbell);
9864                         mptsas_log(mpt, CE_WARN, "MPT Forced Reset, "
9865                             "doorbell: %04x", doorbell);
9866                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
9867                         if (mpt->m_softstate & MPTSAS_SS_MUR_INWATCH)
9868                                 mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
9869                         mpt->m_softstate &= ~(MPTSAS_SS_RESET_INWATCH|
```



```
9870                             MPTSAS_SS_MUR_INWATCH);
9871                         if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
9872                                 mptsas_log(mpt, CE_WARN, "Reset failed"
9873                                     "after fault was detected");
9874                         }
9875                 }
9876

9878                 /*
9879 #endif /* ! codereview */
9880                  * For now, always call mptsas_watchsubr.
9881                  */
9882                 mptsas_watchsubr(mpt);

9884                 if (mpt->m_options & MPTSAS_OPT_PM) {
9885                         mpt->m_busy = 0;
9886                         (void) pm_idle_component(mpt->m_dip, 0);
9887                 }

9889                 mutex_exit(&mpt->m_mutex);
9890         }
9891         rw_exit(&mptsas_global_rwlock);

9893         mutex_enter(&mptsas_global_mutex);
9894         if (mptsas_timeouts_enabled)
9895                 mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
9896         mutex_exit(&mptsas_global_mutex);
9897 }

9899 static void
9900 mptsas_watchsubr(mptsas_t *mpt)
9901 {
9902         int             i;
9903         mptsas_cmd_t    *cmd;
9904         mptsas_target_t *ptgt = NULL;
9905         hrtime_t        timestamp = gethrtime();

9907         ASSERT(MUTEX_HELD(&mpt->m_mutex));

9909         NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

9911 #ifdef MPTSAS_TEST
9912         if (mptsas_enable_untagged) {
9913                 mptsas_test_untagged++;
9914         }
9915 #endif

9917         /*
9918          * Check for commands stuck in active slot
9919          * Account for TM requests, which use the last SMID.
9920          */
9921         for (i = 0; i <= mpt->m_active->m_n_normal; i++) {
9922                 if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9923                         if (cmd->cmd_active_expiration <= timestamp) {
9924                                 if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9925                                         /*
9926                                          * There seems to be a command stuck
9927                                          * in the active slot.  Drain throttle.
9928                                          */
9929                                         mptsas_set_throttle(mpt,
9930                                             cmd->cmd_tgt_addr,
9931                                             DRAIN_THROTTLE);
9932                                 } else if (cmd->cmd_flags &
9933                                     (CFLAG_PASSTHRU | CFLAG_CONFIG |
9934                                     CFLAG_FW_DIAG)) {
9935                                         /*
```

```
9936                                     * passthrough command timeout
9937                                     */
9938                                    cmd->cmd_flags |= (CFLAG_FINISHED |
9939                                        CFLAG_TIMEOUT);
9940                                    cv_broadcast(&mpt->m_passthru_cv);
9941                                    cv_broadcast(&mpt->m_config_cv);
9942                                    cv_broadcast(&mpt->m_fw_diag_cv);
9943                                }
9944                        }
9945                }
9946        }

9948        for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
9949            ptgt = refhash_next(mpt->m_targets, ptgt)) {
9950                /*
9951                 * If we were draining due to a qfull condition,
9952                 * go back to full throttle.
9953                 */
9954                if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9955                    (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9956                    (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9957                        mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9958                        mptsas_restart_hba(mpt);
9959                }

9961                cmd = TAILQ_LAST(&ptgt->m_active_cmdq, mptsas_active_cmdq);
9962                if (cmd == NULL)
9963                        continue;

9965                if (cmd->cmd_active_expiration <= timestamp) {
9966                        /*
9967                         * Earliest command timeout expired. Drain throttle.
9968                         */
9969                        mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);

9971                        /*
9972                         * Check for remaining commands.
9973                         */
9974                        cmd = TAILQ_FIRST(&ptgt->m_active_cmdq);
9975                        if (cmd->cmd_active_expiration > timestamp) {
9976                                /*
9977                                 * Wait for remaining commands to complete or
9978                                 * time out.
9979                                 */
9980                                NDBG23(("command timed out, pending drain"));
9981                                continue;
9982                        }

9984                        /*
9985                         * All command timeouts expired.
9986                         */
9987                        mptsas_log(mpt, CE_NOTE, "Timeout of %d seconds "
9988                            "expired with %d commands on target %d lun %d.",
9989                            cmd->cmd_pkt->pkt_time, ptgt->m_t_ncmds,
9990                            ptgt->m_devhdl, Lun(cmd));

9992                        mptsas_cmd_timeout(mpt, ptgt);
9993                } else if (cmd->cmd_active_expiration <=
9994                    timestamp + (hrtime_t)mptsas_scsi_watchdog_tick * NANOSEC) {
9995                        NDBG23(("pending timeout"));
9996                        mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
9997                }
9998        }
9999 }

10001 /*
```



```
10002  * timeout recovery
10003  */
10004 static void
10005 mptsas_cmd_timeout(mptsas_t *mpt, mptsas_target_t *ptgt)
10006 {
10007         uint16_t        devhdl;
10008         uint64_t        sas_wwn;
10009         uint8_t         phy;
10010         char            wwn_str[MPTSAS_WWN_STRLEN];

10012         devhdl = ptgt->m_devhdl;
10013         sas_wwn = ptgt->m_addr.mta_wwn;
10014         phy = ptgt->m_phynum;
10015         if (sas_wwn == 0) {
10016                 (void) sprintf(wwn_str, "p%x", phy);
10017         } else {
10018                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
10019         }

10021         NDBG29(("mptsas_cmd_timeout: target=%d", devhdl));
10022         mptsas_log(mpt, CE_WARN, "Disconnected command timeout for "
10023             "target %d %s, enclosure %u", devhdl, wwn_str,
10024             ptgt->m_enclosure);

10026         /*
10027          * Abort all outstanding commands on the device.
10028          */
10029         NDBG29(("mptsas_cmd_timeout: device reset"));
10030         if (mptsas_do_scsi_reset(mpt, devhdl) != TRUE) {
10031                 mptsas_log(mpt, CE_WARN, "Target %d reset for command timeout "
10032                     "recovery failed!", devhdl);
10033         }
10034 }

10036 /*
10037  * Device / Hotplug control
10038  */
10039 static int
10040 mptsas_scsi_quiesce(dev_info_t *dip)
10041 {
10042         mptsas_t        *mpt;
10043         scsi_hba_tran_t *tran;

10045         tran = ddi_get_driver_private(dip);
10046         if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
10047                 return (-1);

10049         return (mptsas_quiesce_bus(mpt));
10050 }

10052 static int
10053 mptsas_scsi_unquiesce(dev_info_t *dip)
10054 {
10055         mptsas_t                *mpt;
10056         scsi_hba_tran_t *tran;

10058         tran = ddi_get_driver_private(dip);
10059         if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
10060                 return (-1);

10062         return (mptsas_unquiesce_bus(mpt));
10063 }

10065 static int
10066 mptsas_quiesce_bus(mptsas_t *mpt)
10067 {
```

```
10068         mptsas_target_t *ptgt = NULL;

10070         NDBG28(("mptsas_quiesce_bus"));
10071         mutex_enter(&mpt->m_mutex);

10073         /* Set all the throttles to zero */
10074         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10075             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10076                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10077         }

10079         /* If there are any outstanding commands in the queue */
10080         if (mpt->m_ncmds) {
10081                 mpt->m_softstate |= MPTSAS_SS_DRAINING;
10082                 mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10083                     mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
10084                 if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
10085                         /*
10086                          * Quiesce has been interrupted
10087                          */
10088                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10089                         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10090                             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10091                                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10092                         }
10093                         mptsas_restart_hba(mpt);
10094                         if (mpt->m_quiesce_timeid != 0) {
10095                                 timeout_id_t tid = mpt->m_quiesce_timeid;
10096                                 mpt->m_quiesce_timeid = 0;
10097                                 mutex_exit(&mpt->m_mutex);
10098                                 (void) untimeout(tid);
10099                                 return (-1);
10100                         }
10101                         mutex_exit(&mpt->m_mutex);
10102                         return (-1);
10103                 } else {
10104                         /* Bus has been quiesced */
10105                         ASSERT(mpt->m_quiesce_timeid == 0);
10106                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10107                         mpt->m_softstate |= MPTSAS_SS_QUIESCED;
10108                         mutex_exit(&mpt->m_mutex);
10109                         return (0);
10110                 }
10111         }
10112         /* Bus was not busy - QUIESCED */
10113         mutex_exit(&mpt->m_mutex);

10115         return (0);
10116 }

10118 static int
10119 mptsas_unquiesce_bus(mptsas_t *mpt)
10120 {
10121         mptsas_target_t *ptgt = NULL;

10123         NDBG28(("mptsas_unquiesce_bus"));
10124         mutex_enter(&mpt->m_mutex);
10125         mpt->m_softstate &= ~MPTSAS_SS_QUIESCED;
10126         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10127             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10128                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10129         }
10130         mptsas_restart_hba(mpt);
10131         mutex_exit(&mpt->m_mutex);
10132         return (0);
10133 }
```



```
10135 static void
10136 mptsas_ncmds_checkdrain(void *arg)
10137 {
10138         mptsas_t        *mpt = arg;
10139         mptsas_target_t *ptgt = NULL;

10141         mutex_enter(&mpt->m_mutex);
10142         if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
10143                 mpt->m_quiesce_timeid = 0;
10144                 if (mpt->m_ncmds == 0) {
10145                         /* Command queue has been drained */
10146                         cv_signal(&mpt->m_cv);
10147                 } else {
10148                         /*
10149                          * The throttle may have been reset because
10150                          * of a SCSI bus reset
10151                          */
10152                         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10153                             ptgt = refhash_next(mpt->m_targets, ptgt)) {
10154                                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10155                         }

10157                         mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10158                             mpt, (MPTSAS_QUIESCE_TIMEOUT *
10159                             drv_usectohz(1000000)));
10160                 }
10161         }
10162         mutex_exit(&mpt->m_mutex);
10163 }

10165 /*ARGSUSED*/
10166 static void
10167 mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
10168 {
10169         int     i;
10170         uint8_t *cp = (uchar_t *)cmd->cmd_pkt->pkt_cdbp;
10171         char    buf[128];

10173         buf[0] = '\0';
10174         NDBG25(("?Cmd (0x%p) dump for Target %d Lun %d:\n", (void *)cmd,
10175             Tgt(cmd), Lun(cmd)));
10176         (void) sprintf(&buf[0], "\tcdb=[");
10177         for (i = 0; i < (int)cmd->cmd_cdblen; i++) {
10178                 (void) sprintf(&buf[strlen(buf)], " 0x%x", *cp++);
10179         }
10180         (void) sprintf(&buf[strlen(buf)], " ]");
10181         NDBG25(("?%s\n", buf));
10182         NDBG25(("?pkt_flags=0x%x pkt_statistics=0x%x pkt_state=0x%x\n",
10183             cmd->cmd_pkt->pkt_flags, cmd->cmd_pkt->pkt_statistics,
10184             cmd->cmd_pkt->pkt_state));
10185         NDBG25(("?pkt_scbp=0x%x cmd_flags=0x%x\n", cmd->cmd_pkt->pkt_scbp ?
10186             *(cmd->cmd_pkt->pkt_scbp) : 0, cmd->cmd_flags));
10187 }

10189 static void
10190 mptsas_passthru_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10191     pMpi2SGESimple64_t sgep)
10192 {
10193         uint32_t                sge_flags;
10194         uint32_t                data_size, dataout_size;
10195         ddi_dma_cookie_t        data_cookie;
10196         ddi_dma_cookie_t        dataout_cookie;

10198         data_size = pt->data_size;
10199         dataout_size = pt->dataout_size;
```

```
10200         data_cookie = pt->data_cookie;
10201         dataout_cookie = pt->dataout_cookie;

10203         if (dataout_size) {
10204                 sge_flags = dataout_size |
10205                     ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10206                     MPI2_SGE_FLAGS_END_OF_BUFFER |
10207                     MPI2_SGE_FLAGS_HOST_TO_IOC |
10208                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10209                     MPI2_SGE_FLAGS_SHIFT);
10210                 ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
10211                 ddi_put32(acc_hdl, &sgep->Address.Low,
10212                     (uint32_t)(dataout_cookie.dmac_laddress &
10213                     0xffffffffull));
10214                 ddi_put32(acc_hdl, &sgep->Address.High,
10215                     (uint32_t)(dataout_cookie.dmac_laddress
10216                     >> 32));
10217                 sgep++;
10218         }
10219         sge_flags = data_size;
10220         sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10221             MPI2_SGE_FLAGS_LAST_ELEMENT |
10222             MPI2_SGE_FLAGS_END_OF_BUFFER |
10223             MPI2_SGE_FLAGS_END_OF_LIST |
10224             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10225             MPI2_SGE_FLAGS_SHIFT);
10226         if (pt->direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10227                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
10228                     MPI2_SGE_FLAGS_SHIFT);
10229         } else {
10230                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
10231                     MPI2_SGE_FLAGS_SHIFT);
10232         }
10233         ddi_put32(acc_hdl, &sgep->FlagsLength,
10234             sge_flags);
10235         ddi_put32(acc_hdl, &sgep->Address.Low,
10236             (uint32_t)(data_cookie.dmac_laddress &
10237             0xffffffffull));
10238         ddi_put32(acc_hdl, &sgep->Address.High,
10239             (uint32_t)(data_cookie.dmac_laddress >> 32));
10240 }

10242 static void
10243 mptsas_passthru_ieee_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10244     pMpi2IeeeSgeSimple64_t ieeesgep)
10245 {
10246         uint8_t                 sge_flags;
10247         uint32_t                data_size, dataout_size;
10248         ddi_dma_cookie_t        data_cookie;
10249         ddi_dma_cookie_t        dataout_cookie;

10251         data_size = pt->data_size;
10252         dataout_size = pt->dataout_size;
10253         data_cookie = pt->data_cookie;
10254         dataout_cookie = pt->dataout_cookie;

10256         sge_flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
10257             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
10258         if (dataout_size) {
10259                 ddi_put32(acc_hdl, &ieeesgep->Length, dataout_size);
10260                 ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10261                     (uint32_t)(dataout_cookie.dmac_laddress &
10262                     0xffffffffull));
10263                 ddi_put32(acc_hdl, &ieeesgep->Address.High,
10264                     (uint32_t)(dataout_cookie.dmac_laddress >> 32));
10265                 ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
```



```
10266                 ieeesgep++;
10267         }
10268         sge_flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
10269         ddi_put32(acc_hdl, &ieeesgep->Length, data_size);
10270         ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10271             (uint32_t)(data_cookie.dmac_laddress & 0xffffffffull));
10272         ddi_put32(acc_hdl, &ieeesgep->Address.High,
10273             (uint32_t)(data_cookie.dmac_laddress >> 32));
10274         ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
10275 }

10277 static void
10278 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
10279 {
10280         caddr_t                 memp;
10281         pMPI2RequestHeader_t    request_hdrp;
10282         struct scsi_pkt         *pkt = cmd->cmd_pkt;
10283         mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
10284         uint32_t                request_size;
10285         uint32_t                i;
10286         uint64_t                request_desc = 0;
10287         uint8_t                 desc_type;
10288         uint16_t                SMID;
10289         uint8_t                 *request, function;
10290         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
10291         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

10293         desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

10295         request = pt->request;
10296         request_size = pt->request_size;

10298         SMID = cmd->cmd_slot;

10300         /*
10301          * Store the passthrough message in memory location
10302          * corresponding to our slot number
10303          */
10304         memp = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
10305         request_hdrp = (pMPI2RequestHeader_t)memp;
10306         bzero(memp, mpt->m_req_frame_size);

10308         for (i = 0; i < request_size; i++) {
10309                 bcopy(request + i, memp + i, 1);
10310         }

10312         NDBG15(("mptsas_start_passthru: Func 0x%x, MsgFlags 0x%x, "
10313             "size=%d, in %d, out %d, SMID %d", request_hdrp->Function,
10314             request_hdrp->MsgFlags, request_size,
10315             pt->data_size, pt->dataout_size, SMID));

10317         /*
10318          * Add an SGE, even if the length is zero.
10319          */
10320         if (mpt->m_MPI25 && pt->simple == 0) {
10321                 mptsas_passthru_ieee_sge(acc_hdl, pt,
10322                     (pMpi2IeeeSgeSimple64_t)
10323                     ((uint8_t *)request_hdrp + pt->sgl_offset));
10324         } else {
10325                 mptsas_passthru_sge(acc_hdl, pt,
10326                     (pMpi2SGESimple64_t)
10327                     ((uint8_t *)request_hdrp + pt->sgl_offset));
10328         }

10330         function = request_hdrp->Function;
10331         if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
```

```
10332             (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10333                 pMpi2SCSIIORequest_t    scsi_io_req;
10334                 caddr_t                 arsbuf;
10335                 uint8_t                 ars_size;
10336                 uint32_t                ars_dmaaddrlow;

10338                 NDBG15(("mptsas_start_passthru: Is SCSI IO Req"));
10339                 scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;

10341                 if (cmd->cmd_extrqslen != 0) {
10342                         /*
10343                          * Mapping of the buffer was done in
10344                          * mptsas_do_passthru().
10345                          * Calculate the DMA address with the same offset.
10346                          */
10347                         arsbuf = cmd->cmd_arq_buf;
10348                         ars_size = cmd->cmd_extrqslen;
10349                         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
10350                             ((uintptr_t)arsbuf - (uintptr_t)mpt->m_req_sense)) &
10351                             0xffffffffu;
10352                 } else {
10353                         arsbuf = mpt->m_req_sense +
10354                             (mpt->m_req_sense_size * (SMID-1));
10355                         cmd->cmd_arq_buf = arsbuf;
10356                         ars_size = mpt->m_req_sense_size;
10357                         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
10358                             (mpt->m_req_sense_size * (SMID-1))) &
10359                             0xffffffffu;
10360                 }
10361                 bzero(arsbuf, ars_size);

10363                 ddi_put8(acc_hdl, &scsi_io_req->SenseBufferLength, ars_size);
10364                 ddi_put32(acc_hdl, &scsi_io_req->SenseBufferLowAddress,
10365                     ars_dmaaddrlow);

10367                 /*
10368                  * Put SGE for data and data_out buffer at the end of
10369                  * scsi_io_request message header.(64 bytes in total)
10370                  * Set SGLOffset0 value
10371                  */
10372                 ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
10373                     offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);

10375                 /*
10376                  * Setup descriptor info.  RAID passthrough must use the
10377                  * default request descriptor which is already set, so if this
10378                  * is a SCSI IO request, change the descriptor to SCSI IO.
10379                  */
10380                 if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
10381                         desc_type = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
10382                         request_desc = ((uint64_t)ddi_get16(acc_hdl,
10383                             &scsi_io_req->DevHandle) << 48);
10384                 }
10385                 (void) ddi_dma_sync(mpt->m_dma_req_sense_hdl, 0, 0,
10386                     DDI_DMA_SYNC_FORDEV);
10387         }

10389         /*
10390          * We must wait till the message has been completed before
10391          * beginning the next message so we wait for this one to
10392          * finish.
10393          */
10394         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
10395         request_desc |= (SMID << 16) + desc_type;
10396         cmd->cmd_rfm = NULL;
10397         MPTSAS_START_CMD(mpt, request_desc);
```



```
10398         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
10399             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
10400                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10401         }
10402 }

10404 typedef void (mptsas_pre_f)(mptsas_t *, mptsas_pt_request_t *);
10405 static mptsas_pre_f     mpi_pre_ioc_facts;
10406 static mptsas_pre_f     mpi_pre_port_facts;
10407 static mptsas_pre_f     mpi_pre_fw_download;
10408 static mptsas_pre_f     mpi_pre_fw_25_download;
10409 static mptsas_pre_f     mpi_pre_fw_upload;
10410 static mptsas_pre_f     mpi_pre_fw_25_upload;
10411 static mptsas_pre_f     mpi_pre_sata_passthrough;
10412 static mptsas_pre_f     mpi_pre_smp_passthrough;
10413 static mptsas_pre_f     mpi_pre_config;
10414 static mptsas_pre_f     mpi_pre_sas_io_unit_control;
10415 static mptsas_pre_f     mpi_pre_scsi_io_req;

10417 /*
10418  * Prepare the pt for a SAS2 FW_DOWNLOAD request.
10419  */
10420 static void
10421 mpi_pre_fw_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10422 {
10423         pMpi2FWDownloadTCSGE_t tcsge;
10424         pMpi2FWDownloadRequest req;

10426         /*
10427          * If SAS3, call separate function.
10428          */
10429         if (mpt->m_MPI25) {
10430                 mpi_pre_fw_25_download(mpt, pt);
10431                 return;
10432         }

10434         /*
10435          * User requests should come in with the Transaction
10436          * context element where the SGL will go. Putting the
10437          * SGL after that seems to work, but don't really know
10438          * why. Other drivers tend to create an extra SGL and
10439          * refer to the TCE through that.
10440          */
10441         req = (pMpi2FWDownloadRequest)pt->request;
10442         tcsge = (pMpi2FWDownloadTCSGE_t)&req->SGL;
10443         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10444             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10445                 mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10446         }

10448         pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL) +
10449             sizeof (*tcsge);
10450         if (pt->request_size != pt->sgl_offset)
10451                 NDBG15(("mpi_pre_fw_download(): Incorrect req size, "
10452                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10453                     (int)pt->request_size, (int)pt->sgl_offset,
10454                     (int)pt->dataout_size));
10455         if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10456                 NDBG15(("mpi_pre_fw_download(): Incorrect rep size, "
10457                     "0x%x, should be 0x%x", pt->data_size,
10458                     (int)sizeof (MPI2_FW_DOWNLOAD_REPLY)));
10459 }

10461 /*
10462  * Prepare the pt for a SAS3 FW_DOWNLOAD request.
10463  */
```

```
10464 static void
10465 mpi_pre_fw_25_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10466 {
10467         pMpi2FWDownloadTCSGE_t tcsge;
10468         pMpi2FWDownloadRequest req2;
10469         pMpi25FWDownloadRequest req25;

10471         /*
10472          * User requests should come in with the Transaction
10473          * context element where the SGL will go. The new firmware
10474          * Doesn't use TCE and has space in the main request for
10475          * this information. So move to the right place.
10476          */
10477         req2 = (pMpi2FWDownloadRequest)pt->request;
10478         req25 = (pMpi25FWDownloadRequest)pt->request;
10479         tcsge = (pMpi2FWDownloadTCSGE_t)&req2->SGL;
10480         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10481             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10482                 mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10483         }
10484         req25->ImageOffset = tcsge->ImageOffset;
10485         req25->ImageSize = tcsge->ImageSize;

10487         pt->sgl_offset = offsetof(MPI25_FW_DOWNLOAD_REQUEST, SGL);
10488         if (pt->request_size != pt->sgl_offset)
10489                 NDBG15(("mpi_pre_fw_25_download(): Incorrect req size, "
10490                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10491                     pt->request_size, pt->sgl_offset,
10492                     pt->dataout_size));
10493         if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10494                 NDBG15(("mpi_pre_fw_25_download(): Incorrect rep size, "
10495                     "0x%x, should be 0x%x", pt->data_size,
10496                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10497 }

10499 /*
10500  * Prepare the pt for a SAS2 FW_UPLOAD request.
10501  */
10502 static void
10503 mpi_pre_fw_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10504 {
10505         pMpi2FWUploadTCSGE_t tcsge;
10506         pMpi2FWUploadRequest_t req;

10508         /*
10509          * If SAS3, call separate function.
10510          */
10511         if (mpt->m_MPI25) {
10512                 mpi_pre_fw_25_upload(mpt, pt);
10513                 return;
10514         }

10516         /*
10517          * User requests should come in with the Transaction
10518          * context element where the SGL will go. Putting the
10519          * SGL after that seems to work, but don't really know
10520          * why. Other drivers tend to create an extra SGL and
10521          * refer to the TCE through that.
10522          */
10523         req = (pMpi2FWUploadRequest_t)pt->request;
10524         tcsge = (pMpi2FWUploadTCSGE_t)&req->SGL;
10525         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10526             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10527                 mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10528         }
```



```
10530         pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL) +
10531             sizeof (*tcsge);
10532         if (pt->request_size != pt->sgl_offset)
10533                 NDBG15(("mpi_pre_fw_upload(): Incorrect req size, "
10534                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10535                     pt->request_size, pt->sgl_offset,
10536                     pt->dataout_size));
10537         if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10538                 NDBG15(("mpi_pre_fw_upload(): Incorrect rep size, "
10539                     "0x%x, should be 0x%x", pt->data_size,
10540                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10541 }

10543 /*
10544  * Prepare the pt a SAS3 FW_UPLOAD request.
10545  */
10546 static void
10547 mpi_pre_fw_25_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10548 {
10549         pMpi2FWUploadTCSGE_t tcsge;
10550         pMpi2FWUploadRequest_t req2;
10551         pMpi25FWUploadRequest_t req25;

10553         /*
10554          * User requests should come in with the Transaction
10555          * context element where the SGL will go. The new firmware
10556          * Doesn't use TCE and has space in the main request for
10557          * this information. So move to the right place.
10558          */
10559         req2 = (pMpi2FWUploadRequest_t)pt->request;
10560         req25 = (pMpi25FWUploadRequest_t)pt->request;
10561         tcsge = (pMpi2FWUploadTCSGE_t)&req2->SGL;
10562         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10563             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10564                 mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10565         }
10566         req25->ImageOffset = tcsge->ImageOffset;
10567         req25->ImageSize = tcsge->ImageSize;

10569         pt->sgl_offset = offsetof(MPI25_FW_UPLOAD_REQUEST, SGL);
10570         if (pt->request_size != pt->sgl_offset)
10571                 NDBG15(("mpi_pre_fw_25_upload(): Incorrect req size, "
10572                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10573                     pt->request_size, pt->sgl_offset,
10574                     pt->dataout_size));
10575         if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10576                 NDBG15(("mpi_pre_fw_25_upload(): Incorrect rep size, "
10577                     "0x%x, should be 0x%x", pt->data_size,
10578                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10579 }

10581 /*
10582  * Prepare the pt for an IOC_FACTS request.
10583  */
10584 static void
10585 mpi_pre_ioc_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10586 {
10587 #ifndef __lock_lint
10588         _NOTE(ARGUNUSED(mpt))
10589 #endif
10590         if (pt->request_size != sizeof (MPI2_IOC_FACTS_REQUEST))
10591                 NDBG15(("mpi_pre_ioc_facts(): Incorrect req size, "
10592                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10593                     pt->request_size,
10594                     (int)sizeof (MPI2_IOC_FACTS_REQUEST),
10595                     pt->dataout_size));
```

```
10596         if (pt->data_size != sizeof (MPI2_IOC_FACTS_REPLY))
10597                 NDBG15(("mpi_pre_ioc_facts(): Incorrect rep size, "
10598                     "0x%x, should be 0x%x", pt->data_size,
10599                     (int)sizeof (MPI2_IOC_FACTS_REPLY)));
10600         pt->sgl_offset = (uint16_t)pt->request_size;
10601 }

10603 /*
10604  * Prepare the pt for a PORT_FACTS request.
10605  */
10606 static void
10607 mpi_pre_port_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10608 {
10609 #ifndef __lock_lint
10610         _NOTE(ARGUNUSED(mpt))
10611 #endif
10612         if (pt->request_size != sizeof (MPI2_PORT_FACTS_REQUEST))
10613                 NDBG15(("mpi_pre_port_facts(): Incorrect req size, "
10614                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10615                     pt->request_size,
10616                     (int)sizeof (MPI2_PORT_FACTS_REQUEST),
10617                     pt->dataout_size));
10618         if (pt->data_size != sizeof (MPI2_PORT_FACTS_REPLY))
10619                 NDBG15(("mpi_pre_port_facts(): Incorrect rep size, "
10620                     "0x%x, should be 0x%x", pt->data_size,
10621                     (int)sizeof (MPI2_PORT_FACTS_REPLY)));
10622         pt->sgl_offset = (uint16_t)pt->request_size;
10623 }

10625 /*
10626  * Prepare pt for a SATA_PASSTHROUGH request.
10627  */
10628 static void
10629 mpi_pre_sata_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10630 {
10631 #ifndef __lock_lint
10632         _NOTE(ARGUNUSED(mpt))
10633 #endif
10634         pt->sgl_offset = offsetof(MPI2_SATA_PASSTHROUGH_REQUEST, SGL);
10635         if (pt->request_size != pt->sgl_offset)
10636                 NDBG15(("mpi_pre_sata_passthrough(): Incorrect req size, "
10637                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10638                     pt->request_size, pt->sgl_offset,
10639                     pt->dataout_size));
10640         if (pt->data_size != sizeof (MPI2_SATA_PASSTHROUGH_REPLY))
10641                 NDBG15(("mpi_pre_sata_passthrough(): Incorrect rep size, "
10642                     "0x%x, should be 0x%x", pt->data_size,
10643                     (int)sizeof (MPI2_SATA_PASSTHROUGH_REPLY)));
10644 }

10646 static void
10647 mpi_pre_smp_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10648 {
10649 #ifndef __lock_lint
10650         _NOTE(ARGUNUSED(mpt))
10651 #endif
10652         pt->sgl_offset = offsetof(MPI2_SMP_PASSTHROUGH_REQUEST, SGL);
10653         if (pt->request_size != pt->sgl_offset)
10654                 NDBG15(("mpi_pre_smp_passthrough(): Incorrect req size, "
10655                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10656                     pt->request_size, pt->sgl_offset,
10657                     pt->dataout_size));
10658         if (pt->data_size != sizeof (MPI2_SMP_PASSTHROUGH_REPLY))
10659                 NDBG15(("mpi_pre_smp_passthrough(): Incorrect rep size, "
10660                     "0x%x, should be 0x%x", pt->data_size,
10661                     (int)sizeof (MPI2_SMP_PASSTHROUGH_REPLY)));
```



```
10662 }

10664 /*
10665  * Prepare pt for a CONFIG request.
10666  */
10667 static void
10668 mpi_pre_config(mptsas_t *mpt, mptsas_pt_request_t *pt)
10669 {
10670 #ifndef __lock_lint
10671         _NOTE(ARGUNUSED(mpt))
10672 #endif
10673         pt->sgl_offset = offsetof(MPI2_CONFIG_REQUEST, PageBufferSGE);
10674         if (pt->request_size != pt->sgl_offset)
10675                 NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10676                     "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10677                     pt->sgl_offset, pt->dataout_size));
10678         if (pt->data_size != sizeof (MPI2_CONFIG_REPLY))
10679                 NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10680                     "should be 0x%x", pt->data_size,
10681                     (int)sizeof (MPI2_CONFIG_REPLY)));
10682         pt->simple = 1;
10683 }

10685 /*
10686  * Prepare pt for a SCSI_IO_REQ request.
10687  */
10688 static void
10689 mpi_pre_scsi_io_req(mptsas_t *mpt, mptsas_pt_request_t *pt)
10690 {
10691 #ifndef __lock_lint
10692         _NOTE(ARGUNUSED(mpt))
10693 #endif
10694         pt->sgl_offset = offsetof(MPI2_SCSI_IO_REQUEST, SGL);
10695         if (pt->request_size != pt->sgl_offset)
10696                 NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10697                     "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10698                     pt->sgl_offset,
10699                     pt->dataout_size));
10700         if (pt->data_size != sizeof (MPI2_SCSI_IO_REPLY))
10701                 NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10702                     "should be 0x%x", pt->data_size,
10703                     (int)sizeof (MPI2_SCSI_IO_REPLY)));
10704 }

10706 /*
10707  * Prepare the mptsas_cmd for a SAS_IO_UNIT_CONTROL request.
10708  */
10709 static void
10710 mpi_pre_sas_io_unit_control(mptsas_t *mpt, mptsas_pt_request_t *pt)
10711 {
10712 #ifndef __lock_lint
10713         _NOTE(ARGUNUSED(mpt))
10714 #endif
10715         pt->sgl_offset = (uint16_t)pt->request_size;
10716 }

10718 /*
10719  * A set of functions to prepare an mptsas_cmd for the various
10720  * supported requests.
10721  */
10722 static struct mptsas_func {
10723         U8              Function;
10724         char            *Name;
10725         mptsas_pre_f    *f_pre;
10726 } mptsas_func_list[] = {
10727         { MPI2_FUNCTION_IOC_FACTS, "IOC_FACTS",         mpi_pre_ioc_facts },
```

```
10728         { MPI2_FUNCTION_PORT_FACTS, "PORT_FACTS",       mpi_pre_port_facts },
10729         { MPI2_FUNCTION_FW_DOWNLOAD, "FW_DOWNLOAD",     mpi_pre_fw_download },
10730         { MPI2_FUNCTION_FW_UPLOAD, "FW_UPLOAD",         mpi_pre_fw_upload },
10731         { MPI2_FUNCTION_SATA_PASSTHROUGH, "SATA_PASSTHROUGH",
10732             mpi_pre_sata_passthrough },
10733         { MPI2_FUNCTION_SMP_PASSTHROUGH, "SMP_PASSTHROUGH",
10734             mpi_pre_smp_passthrough},
10735         { MPI2_FUNCTION_SCSI_IO_REQUEST, "SCSI_IO_REQUEST",
10736             mpi_pre_scsi_io_req},
10737         { MPI2_FUNCTION_CONFIG, "CONFIG",               mpi_pre_config},
10738         { MPI2_FUNCTION_SAS_IO_UNIT_CONTROL, "SAS_IO_UNIT_CONTROL",
10739             mpi_pre_sas_io_unit_control },
10740         { 0xFF, NULL,                           NULL } /* list end */
10741 };

10743 static void
10744 mptsas_prep_sgl_offset(mptsas_t *mpt, mptsas_pt_request_t *pt)
10745 {
10746         pMPI2RequestHeader_t    hdr;
10747         struct mptsas_func      *f;

10749         hdr = (pMPI2RequestHeader_t)pt->request;

10751         for (f = mptsas_func_list; f->f_pre != NULL; f++) {
10752                 if (hdr->Function == f->Function) {
10753                         f->f_pre(mpt, pt);
10754                         NDBG15(("mptsas_prep_sgl_offset: Function %s,"
10755                             " sgl_offset 0x%x", f->Name,
10756                             pt->sgl_offset));
10757                         return;
10758                 }
10759         }
10760         NDBG15(("mptsas_prep_sgl_offset: Unknown Function 0x%02x,"
10761             " returning req_size 0x%x for sgl_offset",
10762             hdr->Function, pt->request_size));
10763         pt->sgl_offset = (uint16_t)pt->request_size;
10764 }


10767 static int
10768 mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
10769     uint8_t *data, uint32_t request_size, uint32_t reply_size,
10770     uint32_t data_size, uint32_t direction, uint8_t *dataout,
10771     uint32_t dataout_size, short timeout, int mode)
10772 {
10773         mptsas_pt_request_t             pt;
10774         mptsas_dma_alloc_state_t        data_dma_state;
10775         mptsas_dma_alloc_state_t        dataout_dma_state;
10776         caddr_t                         memp;
10777         mptsas_cmd_t                    *cmd = NULL;
10778         struct scsi_pkt                 *pkt;
10779         uint32_t                        reply_len = 0, sense_len = 0;
10780         pMPI2RequestHeader_t            request_hdrp;
10781         pMPI2RequestHeader_t            request_msg;
10782         pMPI2DefaultReply_t             reply_msg;
10783         Mpi2SCSIIOReply_t               rep_msg;
10784         int                             rvalue;
10785         int                             i, status = 0, pt_flags = 0, rv = 0;
10786         uint8_t                         function;

10788         ASSERT(mutex_owned(&mpt->m_mutex));

10790         reply_msg = (pMPI2DefaultReply_t)(&rep_msg);
10791         bzero(reply_msg, sizeof (MPI2_DEFAULT_REPLY));
10792         request_msg = kmem_zalloc(request_size, KM_SLEEP);
```



```
10794         mutex_exit(&mpt->m_mutex);
10795         /*
10796          * copy in the request buffer since it could be used by
10797          * another thread when the pt request into waitq
10798          */
10799         if (ddi_copyin(request, request_msg, request_size, mode)) {
10800                 mutex_enter(&mpt->m_mutex);
10801                 status = EFAULT;
10802                 mptsas_log(mpt, CE_WARN, "failed to copy request data");
10803                 goto out;
10804         }
10805         NDBG27(("mptsas_do_passthru: mode 0x%x, size 0x%x, Func 0x%x",
10806             mode, request_size, request_msg->Function));
10807         mutex_enter(&mpt->m_mutex);

10809         function = request_msg->Function;
10810         if (function == MPI2_FUNCTION_SCSI_TASK_MGMT) {
10811                 pMpi2SCSITaskManagementRequest_t        task;
10812                 task = (pMpi2SCSITaskManagementRequest_t)request_msg;
10813                 mptsas_setup_bus_reset_delay(mpt);
10814                 rv = mptsas_ioc_task_management(mpt, task->TaskType,
10815                     task->DevHandle, (int)task->LUN[1], reply, reply_size,
10816                     mode);

10818                 if (rv != TRUE) {
10819                         status = EIO;
10820                         mptsas_log(mpt, CE_WARN, "task management failed");
10821                 }
10822                 goto out;
10823         }

10825         if (data_size != 0) {
10826                 data_dma_state.size = data_size;
10827                 if (mptsas_dma_alloc(mpt, &data_dma_state) != DDI_SUCCESS) {
10828                         status = ENOMEM;
10829                         mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10830                             "resource");
10831                         goto out;
10832                 }
10833                 pt_flags |= MPTSAS_DATA_ALLOCATED;
10834                 if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10835                         mutex_exit(&mpt->m_mutex);
10836                         for (i = 0; i < data_size; i++) {
10837                                 if (ddi_copyin(data + i, (uint8_t *)
10838                                     data_dma_state.memp + i, 1, mode)) {
10839                                         mutex_enter(&mpt->m_mutex);
10840                                         status = EFAULT;
10841                                         mptsas_log(mpt, CE_WARN, "failed to "
10842                                             "copy read data");
10843                                         goto out;
10844                                 }
10845                         }
10846                         mutex_enter(&mpt->m_mutex);
10847                 }
10848         } else {
10849                 bzero(&data_dma_state, sizeof (data_dma_state));
10850         }

10852         if (dataout_size != 0) {
10853                 dataout_dma_state.size = dataout_size;
10854                 if (mptsas_dma_alloc(mpt, &dataout_dma_state) != DDI_SUCCESS) {
10855                         status = ENOMEM;
10856                         mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10857                             "resource");
10858                         goto out;
10859                 }
```

```
10860                 pt_flags |= MPTSAS_DATAOUT_ALLOCATED;
10861                 mutex_exit(&mpt->m_mutex);
10862                 for (i = 0; i < dataout_size; i++) {
10863                         if (ddi_copyin(dataout + i, (uint8_t *)
10864                             dataout_dma_state.memp + i, 1, mode)) {
10865                             mutex_enter(&mpt->m_mutex);
10866                             mptsas_log(mpt, CE_WARN, "failed to copy out"
10867                                 " data");
10868                             status = EFAULT;
10869                             goto out;
10870                         }
10871                 }
10872                 mutex_enter(&mpt->m_mutex);
10873         } else {
10874                 bzero(&dataout_dma_state, sizeof (dataout_dma_state));
10875         }

10877         if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
10878                 status = EAGAIN;
10879                 mptsas_log(mpt, CE_NOTE, "event ack command pool is full");
10880                 goto out;
10881         }
10882         pt_flags |= MPTSAS_REQUEST_POOL_CMD;

10884         bzero((caddr_t)cmd, sizeof (*cmd));
10885         bzero((caddr_t)pkt, scsi_pkt_size());
10886         bzero((caddr_t)&pt, sizeof (pt));

10888         cmd->ioc_cmd_slot = (uint32_t)(rvalue);

10890         pt.request = (uint8_t *)request_msg;
10891         pt.direction = direction;
10892         pt.simple = 0;
10893         pt.request_size = request_size;
10894         pt.data_size = data_size;
10895         pt.dataout_size = dataout_size;
10896         pt.data_cookie = data_dma_state.cookie;
10897         pt.dataout_cookie = dataout_dma_state.cookie;
10898         mptsas_prep_sgl_offset(mpt, &pt);

10900         /*
10901          * Form a blank cmd/pkt to store the acknowledgement message
10902          */
10903         pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
10904         pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
10905         pkt->pkt_ha_private     = (opaque_t)&pt;
10906         pkt->pkt_flags          = FLAG_HEAD;
10907         pkt->pkt_time           = timeout;
10908         cmd->cmd_pkt            = pkt;
10909         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_PASSTHRU;

10911         if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10912             (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10913                 uint8_t                 com, cdb_group_id;
10914                 boolean_t               ret;

10916                 pkt->pkt_cdbp = ((pMpi2SCSIIORequest_t)request_msg)->CDB.CDB32;
10917                 com = pkt->pkt_cdbp[0];
10918                 cdb_group_id = CDB_GROUPID(com);
10919                 switch (cdb_group_id) {
10920                 case CDB_GROUPID_0: cmd->cmd_cdblen = CDB_GROUP0; break;
10921                 case CDB_GROUPID_1: cmd->cmd_cdblen = CDB_GROUP1; break;
10922                 case CDB_GROUPID_2: cmd->cmd_cdblen = CDB_GROUP2; break;
10923                 case CDB_GROUPID_4: cmd->cmd_cdblen = CDB_GROUP4; break;
10924                 case CDB_GROUPID_5: cmd->cmd_cdblen = CDB_GROUP5; break;
10925                 default:
```



```
10926                         NDBG27(("mptsas_do_passthru: SCSI_IO, reserved "
10927                             "CDBGROUP 0x%x requested!", cdb_group_id));
10928                         break;
10929                 }

10931                 reply_len = sizeof (MPI2_SCSI_IO_REPLY);
10932                 sense_len = reply_size - reply_len;
10933                 ret = mptsas_cmdarqsize(mpt, cmd, sense_len, KM_SLEEP);
10934                 VERIFY(ret == B_TRUE);
10935         } else {
10936                 reply_len = reply_size;
10937                 sense_len = 0;
10938         }

10940         NDBG27(("mptsas_do_passthru: %s, dsz 0x%x, dosz 0x%x, replen 0x%x, "
10941             "snslen 0x%x",
10942             (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE)?"Write":"Read",
10943             data_size, dataout_size, reply_len, sense_len));

10945         /*
10946          * Save the command in a slot
10947          */
10948         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
10949                 /*
10950                  * Once passthru command get slot, set cmd_flags
10951                  * CFLAG_PREPARED.
10952                  */
10953                 cmd->cmd_flags |= CFLAG_PREPARED;
10954                 mptsas_start_passthru(mpt, cmd);
10955         } else {
10956                 mptsas_waitq_add(mpt, cmd);
10957         }

10959         while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
10960                 cv_wait(&mpt->m_passthru_cv, &mpt->m_mutex);
10961         }

10963         NDBG27(("mptsas_do_passthru: Cmd complete, flags 0x%x, rfm 0x%x "
10964             "pktreason 0x%x", cmd->cmd_flags, cmd->cmd_rfm,
10965             pkt->pkt_reason));

10967         if (cmd->cmd_flags & CFLAG_PREPARED) {
10968                 memp = mpt->m_req_frame + (mpt->m_req_frame_size *
10969                     cmd->cmd_slot);
10970                 request_hdrp = (pMPI2RequestHeader_t)memp;
10971         }

10973         if (cmd->cmd_flags & CFLAG_TIMEOUT) {
10974                 status = ETIMEDOUT;
10975                 mptsas_log(mpt, CE_WARN, "passthrough command timeout");
10976                 pt_flags |= MPTSAS_CMD_TIMEOUT;
10977                 goto out;
10978         }

10980         if (cmd->cmd_rfm) {
10981                 /*
10982                  * cmd_rfm is zero means the command reply is a CONTEXT
10983                  * reply and no PCI Write to post the free reply SMFA
10984                  * because no reply message frame is used.
10985                  * cmd_rfm is non-zero means the reply is a ADDRESS
10986                  * reply and reply message frame is used.
10987                  */
10988                 pt_flags |= MPTSAS_ADDRESS_REPLY;
10989                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
10990                     DDI_DMA_SYNC_FORCPU);
10991                 reply_msg = (pMPI2DefaultReply_t)
```

```
10992                     (mpt->m_reply_frame + (cmd->cmd_rfm -
10993                     (mpt->m_reply_frame_dma_addr & 0xffffffffu)));
10994         }
10995
10996         mptsas_fma_check(mpt, cmd);
10997         if (pkt->pkt_reason == CMD_TRAN_ERR) {
10998                 status = EAGAIN;
10999                 mptsas_log(mpt, CE_WARN, "passthru fma error");
11000                 goto out;
11001         }
11002         if (pkt->pkt_reason == CMD_RESET) {
11003                 status = EAGAIN;
11004                 mptsas_log(mpt, CE_WARN, "ioc reset abort passthru");
11005                 goto out;
11006         }
11007
11008         if (pkt->pkt_reason == CMD_INCOMPLETE) {
11009                 status = EIO;
11010                 mptsas_log(mpt, CE_WARN, "passthrough command incomplete");
11011                 goto out;
11012         }
11013
11014         mutex_exit(&mpt->m_mutex);
11015         if (cmd->cmd_flags & CFLAG_PREPARED) {
11016                 function = request_hdrp->Function;
11017                 if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
11018                     (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
11019                         reply_len = sizeof (MPI2_SCSI_IO_REPLY);
11020                         sense_len = cmd->cmd_extrqslen ?
11021                             min(sense_len, cmd->cmd_extrqslen) :
11022                             min(sense_len, cmd->cmd_rqslen);
11023                 } else {
11024                         reply_len = reply_size;
11025                         sense_len = 0;
11026                 }
11027
11028                 for (i = 0; i < reply_len; i++) {
11029                         if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
11030                             mode)) {
11031                                 mutex_enter(&mpt->m_mutex);
11032                                 status = EFAULT;
11033                                 mptsas_log(mpt, CE_WARN, "failed to copy out "
11034                                     "reply data");
11035                                 goto out;
11036                         }
11037                 }
11038                 for (i = 0; i < sense_len; i++) {
11039                         if (ddi_copyout((uint8_t *)request_hdrp + 64 + i,
11040                             reply + reply_len + i, 1, mode)) {
11041                                 mutex_enter(&mpt->m_mutex);
11042                                 status = EFAULT;
11043                                 mptsas_log(mpt, CE_WARN, "failed to copy out "
11044                                     "sense data");
11045                                 goto out;
11046                         }
11047                 }
11048         }
11049
11050         if (data_size) {
11051                 if (direction != MPTSAS_PASS_THRU_DIRECTION_WRITE) {
11052                         (void) ddi_dma_sync(data_dma_state.handle, 0, 0,
11053                             DDI_DMA_SYNC_FORCPU);
11054                         for (i = 0; i < data_size; i++) {
11055                                 if (ddi_copyout((uint8_t *)(
11056                                     data_dma_state.memp + i), data + i,  1,
11057                                     mode)) {
```



```
11058                                         mutex_enter(&mpt->m_mutex);
11059                                         status = EFAULT;
11060                                         mptsas_log(mpt, CE_WARN, "failed to "
11061                                             "copy out the reply data");
11062                                         goto out;
11063                                 }
11064                         }
11065                 }
11066         }
11067         mutex_enter(&mpt->m_mutex);
11068 out:
11069         /*
11070          * Put the reply frame back on the free queue, increment the free
11071          * index, and write the new index to the free index register.  But only
11072          * if this reply is an ADDRESS reply.
11073          */
11074         if (pt_flags & MPTSAS_ADDRESS_REPLY) {
11075                 ddi_put32(mpt->m_acc_free_queue_hdl,
11076                     &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
11077                     cmd->cmd_rfm);
11078                 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11079                     DDI_DMA_SYNC_FORDEV);
11080                 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11081                         mpt->m_free_index = 0;
11082                 }
11083                 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11084                     mpt->m_free_index);
11085         }
11086         if (cmd) {
11087                 if (cmd->cmd_extrqslen != 0) {
11088                         rmfree(mpt->m_erqsense_map, cmd->cmd_extrqschunks,
11089                             cmd->cmd_extrqsidx + 1);
11090                 }
11091                 if (cmd->cmd_flags & CFLAG_PREPARED) {
11092                         mptsas_remove_cmd(mpt, cmd);
11093                         pt_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11094                 }
11095         }
11096         if (pt_flags & MPTSAS_REQUEST_POOL_CMD)
11097                 mptsas_return_to_pool(mpt, cmd);
11098         if (pt_flags & MPTSAS_DATA_ALLOCATED) {
11099                 if (mptsas_check_dma_handle(data_dma_state.handle) !=
11100                     DDI_SUCCESS) {
11101                         ddi_fm_service_impact(mpt->m_dip,
11102                             DDI_SERVICE_UNAFFECTED);
11103                         status = EFAULT;
11104                 }
11105                 mptsas_dma_free(&data_dma_state);
11106         }
11107         if (pt_flags & MPTSAS_DATAOUT_ALLOCATED) {
11108                 if (mptsas_check_dma_handle(dataout_dma_state.handle) !=
11109                     DDI_SUCCESS) {
11110                         ddi_fm_service_impact(mpt->m_dip,
11111                             DDI_SERVICE_UNAFFECTED);
11112                         status = EFAULT;
11113                 }
11114                 mptsas_dma_free(&dataout_dma_state);
11115         }
11116         if (pt_flags & MPTSAS_CMD_TIMEOUT) {
11117                 if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
11118                         mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
11119                 }
11120         }
11121         if (request_msg)
11122                 kmem_free(request_msg, request_size);
11123         NDBG27(("mptsas_do_passthru: Done status 0x%x", status));
```

```
11125         return (status);
11126 }

11128 static int
11129 mptsas_pass_thru(mptsas_t *mpt, mptsas_pass_thru_t *data, int mode)
11130 {
11131         /*
11132          * If timeout is 0, set timeout to default of 60 seconds.
11133          */
11134         if (data->Timeout == 0) {
11135                 data->Timeout = MPTSAS_PASS_THRU_TIME_DEFAULT;
11136         }

11138         if (((data->DataSize == 0) &&
11139             (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_NONE)) ||
11140             ((data->DataSize != 0) &&
11141             ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_READ) ||
11142             (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_WRITE) ||
11143             ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) &&
11144             (data->DataOutSize != 0))))) {
11145                 if (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) {
11146                         data->DataDirection = MPTSAS_PASS_THRU_DIRECTION_READ;
11147                 } else {
11148                         data->DataOutSize = 0;
11149                 }
11150                 /*
11151                  * Send passthru request messages
11152                  */
11153                 return (mptsas_do_passthru(mpt,
11154                     (uint8_t *)((uintptr_t)data->PtrRequest),
11155                     (uint8_t *)((uintptr_t)data->PtrReply),
11156                     (uint8_t *)((uintptr_t)data->PtrData),
11157                     data->RequestSize, data->ReplySize,
11158                     data->DataSize, data->DataDirection,
11159                     (uint8_t *)((uintptr_t)data->PtrDataOut),
11160                     data->DataOutSize, data->Timeout, mode));
11161         } else {
11162                 return (EINVAL);
11163         }
11164 }

11166 static uint8_t
11167 mptsas_get_fw_diag_buffer_number(mptsas_t *mpt, uint32_t unique_id)
11168 {
11169         uint8_t index;

11171         for (index = 0; index < MPI2_DIAG_BUF_TYPE_COUNT; index++) {
11172                 if (mpt->m_fw_diag_buffer_list[index].unique_id == unique_id) {
11173                         return (index);
11174                 }
11175         }

11177         return (MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND);
11178 }

11180 static void
11181 mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
11182 {
11183         pMpi2DiagBufferPostRequest_t    pDiag_post_msg;
11184         pMpi2DiagReleaseRequest_t       pDiag_release_msg;
11185         struct scsi_pkt                 *pkt = cmd->cmd_pkt;
11186         mptsas_diag_request_t           *diag = pkt->pkt_ha_private;
11187         uint32_t                        i;
11188         uint64_t                        request_desc;
```



```
11190         ASSERT(mutex_owned(&mpt->m_mutex));

11192         /*
11193          * Form the diag message depending on the post or release function.
11194          */
11195         if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
11196                 pDiag_post_msg = (pMpi2DiagBufferPostRequest_t)
11197                     (mpt->m_req_frame + (mpt->m_req_frame_size *
11198                     cmd->cmd_slot));
11199                 bzero(pDiag_post_msg, mpt->m_req_frame_size);
11200                 ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
11201                     diag->function);
11202                 ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
11203                     diag->pBuffer->buffer_type);
11204                 ddi_put8(mpt->m_acc_req_frame_hdl,
11205                     &pDiag_post_msg->ExtendedType,
11206                     diag->pBuffer->extended_type);
11207                 ddi_put32(mpt->m_acc_req_frame_hdl,
11208                     &pDiag_post_msg->BufferLength,
11209                     diag->pBuffer->buffer_data.size);
11210                 for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);
11211                     i++) {
11212                         ddi_put32(mpt->m_acc_req_frame_hdl,
11213                             &pDiag_post_msg->ProductSpecific[i],
11214                             diag->pBuffer->product_specific[i]);
11215                 }
11216                 ddi_put32(mpt->m_acc_req_frame_hdl,
11217                     &pDiag_post_msg->BufferAddress.Low,
11218                     (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11219                     & 0xffffffffull));
11220                 ddi_put32(mpt->m_acc_req_frame_hdl,
11221                     &pDiag_post_msg->BufferAddress.High,
11222                     (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11223                     >> 32));
11224         } else {
11225                 pDiag_release_msg = (pMpi2DiagReleaseRequest_t)
11226                     (mpt->m_req_frame + (mpt->m_req_frame_size *
11227                     cmd->cmd_slot));
11228                 bzero(pDiag_release_msg, mpt->m_req_frame_size);
11229                 ddi_put8(mpt->m_acc_req_frame_hdl,
11230                     &pDiag_release_msg->Function, diag->function);
11231                 ddi_put8(mpt->m_acc_req_frame_hdl,
11232                     &pDiag_release_msg->BufferType,
11233                     diag->pBuffer->buffer_type);
11234         }

11236         /*
11237          * Send the message
11238          */
11239         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
11240             DDI_DMA_SYNC_FORDEV);
11241         request_desc = (cmd->cmd_slot << 16) +
11242             MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
11243         cmd->cmd_rfm = NULL;
11244         MPTSAS_START_CMD(mpt, request_desc);
11245         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
11246             DDI_SUCCESS) ||
11247             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
11248             DDI_SUCCESS)) {
11249                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11250         }
11251 }

11253 static int
11254 mptsas_post_fw_diag_buffer(mptsas_t *mpt,
11255     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code)
```

```
11256 {
11257         mptsas_diag_request_t           diag;
11258         int                             status, slot_num, post_flags = 0;
11259         mptsas_cmd_t                    *cmd = NULL;
11260         struct scsi_pkt                 *pkt;
11261         pMpi2DiagBufferPostReply_t      reply;
11262         uint16_t                        iocstatus;
11263         uint32_t                        iocloginfo, transfer_length;

11265         /*
11266          * If buffer is not enabled, just leave.
11267          */
11268         *return_code = MPTSAS_FW_DIAG_ERROR_POST_FAILED;
11269         if (!pBuffer->enabled) {
11270                 status = DDI_FAILURE;
11271                 goto out;
11272         }

11274         /*
11275          * Clear some flags initially.
11276          */
11277         pBuffer->force_release = FALSE;
11278         pBuffer->valid_data = FALSE;
11279         pBuffer->owned_by_firmware = FALSE;

11281         /*
11282          * Get a cmd buffer from the cmd buffer pool
11283          */
11284         if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11285                 status = DDI_FAILURE;
11286                 mptsas_log(mpt, CE_NOTE, "command pool is full: Post FW Diag");
11287                 goto out;
11288         }
11289         post_flags |= MPTSAS_REQUEST_POOL_CMD;

11291         bzero((caddr_t)cmd, sizeof (*cmd));
11292         bzero((caddr_t)pkt, scsi_pkt_size());

11294         cmd->ioc_cmd_slot = (uint32_t)(slot_num);

11296         diag.pBuffer = pBuffer;
11297         diag.function = MPI2_FUNCTION_DIAG_BUFFER_POST;

11299         /*
11300          * Form a blank cmd/pkt to store the acknowledgement message
11301          */
11302         pkt->pkt_ha_private     = (opaque_t)&diag;
11303         pkt->pkt_flags          = FLAG_HEAD;
11304         pkt->pkt_time           = 60;
11305         cmd->cmd_pkt            = pkt;
11306         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_DIAG;

11308         /*
11309          * Save the command in a slot
11310          */
11311         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11312                 /*
11313                  * Once passthru command get slot, set cmd_flags
11314                  * CFLAG_PREPARED.
11315                  */
11316                 cmd->cmd_flags |= CFLAG_PREPARED;
11317                 mptsas_start_diag(mpt, cmd);
11318         } else {
11319                 mptsas_waitq_add(mpt, cmd);
11320         }
```



```
11322         while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11323                 cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11324         }

11326         if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11327                 status = DDI_FAILURE;
11328                 mptsas_log(mpt, CE_WARN, "Post FW Diag command timeout");
11329                 goto out;
11330         }

11332         /*
11333          * cmd_rfm points to the reply message if a reply was given.  Check the
11334          * IOCStatus to make sure everything went OK with the FW diag request
11335          * and set buffer flags.
11336          */
11337         if (cmd->cmd_rfm) {
11338                 post_flags |= MPTSAS_ADDRESS_REPLY;
11339                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11340                     DDI_DMA_SYNC_FORCPU);
11341                 reply = (pMpi2DiagBufferPostReply_t)(mpt->m_reply_frame +
11342                     (cmd->cmd_rfm -
11343                     (mpt->m_reply_frame_dma_addr & 0xffffffffu)));

11345                 /*
11346                  * Get the reply message data
11347                  */
11348                 iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11349                     &reply->IOCStatus);
11350                 iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11351                     &reply->IOCLogInfo);
11352                 transfer_length = ddi_get32(mpt->m_acc_reply_frame_hdl,
11353                     &reply->TransferLength);

11355                 /*
11356                  * If post failed quit.
11357                  */
11358                 if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
11359                         status = DDI_FAILURE;
11360                         NDBG13(("post FW Diag Buffer failed: IOCStatus=0x%x, "
11361                             "IOCLogInfo=0x%x, TransferLength=0x%x", iocstatus,
11362                             iocloginfo, transfer_length));
11363                         goto out;
11364                 }

11366                 /*
11367                  * Post was successful.
11368                  */
11369                 pBuffer->valid_data = TRUE;
11370                 pBuffer->owned_by_firmware = TRUE;
11371                 *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11372                 status = DDI_SUCCESS;
11373         }

11375 out:
11376         /*
11377          * Put the reply frame back on the free queue, increment the free
11378          * index, and write the new index to the free index register.  But only
11379          * if this reply is an ADDRESS reply.
11380          */
11381         if (post_flags & MPTSAS_ADDRESS_REPLY) {
11382                 ddi_put32(mpt->m_acc_free_queue_hdl,
11383                     &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
11384                     cmd->cmd_rfm);
11385                 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11386                     DDI_DMA_SYNC_FORDEV);
11387                 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
```

```
11388                           mpt->m_free_index = 0;
11389                   }
11390                   ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11391                       mpt->m_free_index);
11392           }
11393           if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11394                   mptsas_remove_cmd(mpt, cmd);
11395                   post_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11396           }
11397           if (post_flags & MPTSAS_REQUEST_POOL_CMD) {
11398                   mptsas_return_to_pool(mpt, cmd);
11399           }

11401           return (status);
11402   }

11404   static int
11405   mptsas_release_fw_diag_buffer(mptsas_t *mpt,
11406       mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
11407       uint32_t diag_type)
11408   {
11409           mptsas_diag_request_t   diag;
11410           int                     status, slot_num, rel_flags = 0;
11411           mptsas_cmd_t            *cmd = NULL;
11412           struct scsi_pkt         *pkt;
11413           pMpi2DiagReleaseReply_t reply;
11414           uint16_t                iocstatus;
11415           uint32_t                iocloginfo;

11417           /*
11418            * If buffer is not enabled, just leave.
11419            */
11420           *return_code = MPTSAS_FW_DIAG_ERROR_RELEASE_FAILED;
11421           if (!pBuffer->enabled) {
11422                   mptsas_log(mpt, CE_NOTE, "This buffer type is not supported "
11423                       "by the IOC");
11424                   status = DDI_FAILURE;
11425                   goto out;
11426           }

11428           /*
11429            * Clear some flags initially.
11430            */
11431           pBuffer->force_release = FALSE;
11432           pBuffer->valid_data = FALSE;
11433           pBuffer->owned_by_firmware = FALSE;

11435           /*
11436            * Get a cmd buffer from the cmd buffer pool
11437            */
11438           if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11439                   status = DDI_FAILURE;
11440                   mptsas_log(mpt, CE_NOTE, "command pool is full: Release FW "
11441                       "Diag");
11442                   goto out;
11443           }
11444           rel_flags |= MPTSAS_REQUEST_POOL_CMD;

11446           bzero((caddr_t)cmd, sizeof (*cmd));
11447           bzero((caddr_t)pkt, scsi_pkt_size());

11449           cmd->ioc_cmd_slot = (uint32_t)(slot_num);

11451           diag.pBuffer = pBuffer;
11452           diag.function = MPI2_FUNCTION_DIAG_RELEASE;
```



```
11454           /*
11455            * Form a blank cmd/pkt to store the acknowledgement message
11456            */
11457           pkt->pkt_ha_private     = (opaque_t)&diag;
11458           pkt->pkt_flags          = FLAG_HEAD;
11459           pkt->pkt_time           = 60;
11460           cmd->cmd_pkt            = pkt;
11461           cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_DIAG;

11463           /*
11464            * Save the command in a slot
11465            */
11466           if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11467                   /*
11468                    * Once passthru command get slot, set cmd_flags
11469                    * CFLAG_PREPARED.
11470                    */
11471                   cmd->cmd_flags |= CFLAG_PREPARED;
11472                   mptsas_start_diag(mpt, cmd);
11473           } else {
11474                   mptsas_waitq_add(mpt, cmd);
11475           }

11477           while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11478                   cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11479           }

11481           if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11482                   status = DDI_FAILURE;
11483                   mptsas_log(mpt, CE_WARN, "Release FW Diag command timeout");
11484                   goto out;
11485           }

11487           /*
11488            * cmd_rfm points to the reply message if a reply was given.  Check the
11489            * IOCStatus to make sure everything went OK with the FW diag request
11490            * and set buffer flags.
11491            */
11492           if (cmd->cmd_rfm) {
11493                   rel_flags |= MPTSAS_ADDRESS_REPLY;
11494                   (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11495                       DDI_DMA_SYNC_FORCPU);
11496                   reply = (pMpi2DiagReleaseReply_t)(mpt->m_reply_frame +
11497                       (cmd->cmd_rfm -
11498                       (mpt->m_reply_frame_dma_addr & 0xffffffffu)));

11500                   /*
11501                    * Get the reply message data
11502                    */
11503                   iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11504                       &reply->IOCStatus);
11505                   iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11506                       &reply->IOCLogInfo);

11508                   /*
11509                    * If release failed quit.
11510                    */
11511                   if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) ||
11512                       pBuffer->owned_by_firmware) {
11513                           status = DDI_FAILURE;
11514                           NDBG13(("release FW Diag Buffer failed: "
11515                               "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
11516                               iocloginfo));
11517                           goto out;
11518                   }
```

```
11520                 /*
11521                  * Release was successful.
11522                  */
11523                 *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11524                 status = DDI_SUCCESS;

11526                 /*
11527                  * If this was for an UNREGISTER diag type command, clear the
11528                  * unique ID.
11529                  */
11530                 if (diag_type == MPTSAS_FW_DIAG_TYPE_UNREGISTER) {
11531                       pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11532                 }
11533         }

11535 out:
11536         /*
11537          * Put the reply frame back on the free queue, increment the free
11538          * index, and write the new index to the free index register.  But only
11539          * if this reply is an ADDRESS reply.
11540          */
11541         if (rel_flags & MPTSAS_ADDRESS_REPLY) {
11542                 ddi_put32(mpt->m_acc_free_queue_hdl,
11543                     &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
11544                     cmd->cmd_rfm);
11545                 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11546                     DDI_DMA_SYNC_FORDEV);
11547                 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11548                       mpt->m_free_index = 0;
11549                 }
11550                 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11551                     mpt->m_free_index);
11552         }
11553         if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11554                 mptsas_remove_cmd(mpt, cmd);
11555                 rel_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11556         }
11557         if (rel_flags & MPTSAS_REQUEST_POOL_CMD) {
11558                 mptsas_return_to_pool(mpt, cmd);
11559         }

11561         return (status);
11562 }

11564 static int
11565 mptsas_diag_register(mptsas_t *mpt, mptsas_fw_diag_register_t *diag_register,
11566     uint32_t *return_code)
11567 {
11568         mptsas_fw_diagnostic_buffer_t   *pBuffer;
11569         uint8_t                         extended_type, buffer_type, i;
11570         uint32_t                        buffer_size;
11571         uint32_t                        unique_id;
11572         int                             status;

11574         ASSERT(mutex_owned(&mpt->m_mutex));

11576         extended_type = diag_register->ExtendedType;
11577         buffer_type = diag_register->BufferType;
11578         buffer_size = diag_register->RequestedBufferSize;
11579         unique_id = diag_register->UniqueId;

11581         /*
11582          * Check for valid buffer type
11583          */
11584         if (buffer_type >= MPI2_DIAG_BUF_TYPE_COUNT) {
11585                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
```



```
11586                 return (DDI_FAILURE);
11587         }

11589         /*
11590          * Get the current buffer and look up the unique ID.  The unique ID
11591          * should not be found.  If it is, the ID is already in use.
11592          */
11593         i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11594         pBuffer = &mpt->m_fw_diag_buffer_list[buffer_type];
11595         if (i != MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11596                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11597                 return (DDI_FAILURE);
11598         }

11600         /*
11601          * The buffer's unique ID should not be registered yet, and the given
11602          * unique ID cannot be 0.
11603          */
11604         if ((pBuffer->unique_id != MPTSAS_FW_DIAG_INVALID_UID) ||
11605             (unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11606                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11607                 return (DDI_FAILURE);
11608         }

11610         /*
11611          * If this buffer is already posted as immediate, just change owner.
11612          */
11613         if (pBuffer->immediate && pBuffer->owned_by_firmware &&
11614             (pBuffer->unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11615                 pBuffer->immediate = FALSE;
11616                 pBuffer->unique_id = unique_id;
11617                 return (DDI_SUCCESS);
11618         }

11620         /*
11621          * Post a new buffer after checking if it's enabled.  The DMA buffer
11622          * that is allocated will be contiguous (sgl_len = 1).
11623          */
11624         if (!pBuffer->enabled) {
11625                 *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11626                 return (DDI_FAILURE);
11627         }
11628         bzero(&pBuffer->buffer_data, sizeof (mptsas_dma_alloc_state_t));
11629         pBuffer->buffer_data.size = buffer_size;
11630         if (mptsas_dma_alloc(mpt, &pBuffer->buffer_data) != DDI_SUCCESS) {
11631                 mptsas_log(mpt, CE_WARN, "failed to alloc DMA resource for "
11632                     "diag buffer: size = %d bytes", buffer_size);
11633                 *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11634                 return (DDI_FAILURE);
11635         }

11637         /*
11638          * Copy the given info to the diag buffer and post the buffer.
11639          */
11640         pBuffer->buffer_type = buffer_type;
11641         pBuffer->immediate = FALSE;
11642         if (buffer_type == MPI2_DIAG_BUF_TYPE_TRACE) {
11643                 for (i = 0; i < (sizeof (pBuffer->product_specific) / 4);
11644                     i++) {
11645                       pBuffer->product_specific[i] =
11646                           diag_register->ProductSpecific[i];
11647                 }
11648         }
11649         pBuffer->extended_type = extended_type;
11650         pBuffer->unique_id = unique_id;
11651         status = mptsas_post_fw_diag_buffer(mpt, pBuffer, return_code);
```

```
11653         if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11654             DDI_SUCCESS) {
11655                 mptsas_log(mpt, CE_WARN, "Check of DMA handle failed in "
11656                     "mptsas_diag_register.");
11657                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11658                         status = DDI_FAILURE;
11659         }

11661         /*
11662          * In case there was a failure, free the DMA buffer.
11663          */
11664         if (status == DDI_FAILURE) {
11665                 mptsas_dma_free(&pBuffer->buffer_data);
11666         }

11668         return (status);
11669 }

11671 static int
11672 mptsas_diag_unregister(mptsas_t *mpt,
11673     mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code)
11674 {
11675         mptsas_fw_diagnostic_buffer_t   *pBuffer;
11676         uint8_t                         i;
11677         uint32_t                        unique_id;
11678         int                             status;

11680         ASSERT(mutex_owned(&mpt->m_mutex));

11682         unique_id = diag_unregister->UniqueId;

11684         /*
11685          * Get the current buffer and look up the unique ID.  The unique ID
11686          * should be there.
11687          */
11688         i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11689         if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11690                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11691                 return (DDI_FAILURE);
11692         }

11694         pBuffer = &mpt->m_fw_diag_buffer_list[i];

11696         /*
11697          * Try to release the buffer from FW before freeing it.  If release
11698          * fails, don't free the DMA buffer in case FW tries to access it
11699          * later.  If buffer is not owned by firmware, can't release it.
11700          */
11701         if (!pBuffer->owned_by_firmware) {
11702                 status = DDI_SUCCESS;
11703         } else {
11704                 status = mptsas_release_fw_diag_buffer(mpt, pBuffer,
11705                     return_code, MPTSAS_FW_DIAG_TYPE_UNREGISTER);
11706         }

11708         /*
11709          * At this point, return the current status no matter what happens with
11710          * the DMA buffer.
11711          */
11712         pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11713         if (status == DDI_SUCCESS) {
11714                 if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11715                     DDI_SUCCESS) {
11716                         mptsas_log(mpt, CE_WARN, "Check of DMA handle failed "
11717                             "in mptsas_diag_unregister.");
```



```
11718                         ddi_fm_service_impact(mpt->m_dip,
11719                             DDI_SERVICE_UNAFFECTED);
11720                 }
11721                 mptsas_dma_free(&pBuffer->buffer_data);
11722         }

11724         return (status);
11725 }

11727 static int
11728 mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
11729     uint32_t *return_code)
11730 {
11731         mptsas_fw_diagnostic_buffer_t   *pBuffer;
11732         uint8_t                         i;
11733         uint32_t                        unique_id;

11735         ASSERT(mutex_owned(&mpt->m_mutex));

11737         unique_id = diag_query->UniqueId;

11739         /*
11740          * If ID is valid, query on ID.
11741          * If ID is invalid, query on buffer type.
11742          */
11743         if (unique_id == MPTSAS_FW_DIAG_INVALID_UID) {
11744                 i = diag_query->BufferType;
11745                 if (i >= MPI2_DIAG_BUF_TYPE_COUNT) {
11746                         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11747                         return (DDI_FAILURE);
11748                 }
11749         } else {
11750                 i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11751                 if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11752                         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11753                         return (DDI_FAILURE);
11754                 }
11755         }

11757         /*
11758          * Fill query structure with the diag buffer info.
11759          */
11760         pBuffer = &mpt->m_fw_diag_buffer_list[i];
11761         diag_query->BufferType = pBuffer->buffer_type;
11762         diag_query->ExtendedType = pBuffer->extended_type;
11763         if (diag_query->BufferType == MPI2_DIAG_BUF_TYPE_TRACE) {
11764                 for (i = 0; i < (sizeof (diag_query->ProductSpecific) / 4);
11765                     i++) {
11766                         diag_query->ProductSpecific[i] =
11767                             pBuffer->product_specific[i];
11768                 }
11769         }
11770         diag_query->TotalBufferSize = pBuffer->buffer_data.size;
11771         diag_query->DriverAddedBufferSize = 0;
11772         diag_query->UniqueId = pBuffer->unique_id;
11773         diag_query->ApplicationFlags = 0;
11774         diag_query->DiagnosticFlags = 0;

11776         /*
11777          * Set/Clear application flags
11778          */
11779         if (pBuffer->immediate) {
11780                 diag_query->ApplicationFlags &= ~MPTSAS_FW_DIAG_FLAG_APP_OWNED;
11781         } else {
11782                 diag_query->ApplicationFlags |= MPTSAS_FW_DIAG_FLAG_APP_OWNED;
11783         }
```

```
11784         if (pBuffer->valid_data || pBuffer->owned_by_firmware) {
11785                 diag_query->ApplicationFlags |=
11786                     MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11787         } else {
11788                 diag_query->ApplicationFlags &=
11789                     ~MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11790         }
11791         if (pBuffer->owned_by_firmware) {
11792                 diag_query->ApplicationFlags |=
11793                     MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11794         } else {
11795                 diag_query->ApplicationFlags &=
11796                     ~MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11797         }

11799         return (DDI_SUCCESS);
11800 }

11802 static int
11803 mptsas_diag_read_buffer(mptsas_t *mpt,
11804     mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
11805     uint32_t *return_code, int ioctl_mode)
11806 {
11807         mptsas_fw_diagnostic_buffer_t   *pBuffer;
11808         uint8_t                         i, *pData;
11809         uint32_t                        unique_id, byte;
11810         int                             status;

11812         ASSERT(mutex_owned(&mpt->m_mutex));

11814         unique_id = diag_read_buffer->UniqueId;

11816         /*
11817          * Get the current buffer and look up the unique ID.  The unique ID
11818          * should be there.
11819          */
11820         i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11821         if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11822                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11823                 return (DDI_FAILURE);
11824         }

11826         pBuffer = &mpt->m_fw_diag_buffer_list[i];

11828         /*
11829          * Make sure requested read is within limits
11830          */
11831         if (diag_read_buffer->StartingOffset + diag_read_buffer->BytesToRead >
11832             pBuffer->buffer_data.size) {
11833                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11834                 return (DDI_FAILURE);
11835         }

11837         /*
11838          * Copy the requested data from DMA to the diag_read_buffer.  The DMA
11839          * buffer that was allocated is one contiguous buffer.
11840          */
11841         pData = (uint8_t *)(pBuffer->buffer_data.memp +
11842             diag_read_buffer->StartingOffset);
11843         (void) ddi_dma_sync(pBuffer->buffer_data.handle, 0, 0,
11844             DDI_DMA_SYNC_FORCPU);
11845         for (byte = 0; byte < diag_read_buffer->BytesToRead; byte++) {
11846                 if (ddi_copyout(pData + byte, ioctl_buf + byte, 1, ioctl_mode)
11847                     != 0) {
11848                         return (DDI_FAILURE);
11849                 }
```



```
11850         }
11851         diag_read_buffer->Status = 0;

11853         /*
11854          * Set or clear the Force Release flag.
11855          */
11856         if (pBuffer->force_release) {
11857                 diag_read_buffer->Flags |= MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11858         } else {
11859                 diag_read_buffer->Flags &= ~MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11860         }

11862         /*
11863          * If buffer is to be reregistered, make sure it's not already owned by
11864          * firmware first.
11865          */
11866         status = DDI_SUCCESS;
11867         if (!pBuffer->owned_by_firmware) {
11868                 if (diag_read_buffer->Flags & MPTSAS_FW_DIAG_FLAG_REREGISTER) {
11869                         status = mptsas_post_fw_diag_buffer(mpt, pBuffer,
11870                             return_code);
11871                 }
11872         }

11874         return (status);
11875 }

11877 static int
11878 mptsas_diag_release(mptsas_t *mpt, mptsas_fw_diag_release_t *diag_release,
11879     uint32_t *return_code)
11880 {
11881         mptsas_fw_diagnostic_buffer_t   *pBuffer;
11882         uint8_t                         i;
11883         uint32_t                        unique_id;
11884         int                             status;

11886         ASSERT(mutex_owned(&mpt->m_mutex));

11888         unique_id = diag_release->UniqueId;

11890         /*
11891          * Get the current buffer and look up the unique ID.  The unique ID
11892          * should be there.
11893          */
11894         i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11895         if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11896                 *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11897                 return (DDI_FAILURE);
11898         }

11900         pBuffer = &mpt->m_fw_diag_buffer_list[i];

11902         /*
11903          * If buffer is not owned by firmware, it's already been released.
11904          */
11905         if (!pBuffer->owned_by_firmware) {
11906                 *return_code = MPTSAS_FW_DIAG_ERROR_ALREADY_RELEASED;
11907                 return (DDI_FAILURE);
11908         }

11910         /*
11911          * Release the buffer.
11912          */
11913         status = mptsas_release_fw_diag_buffer(mpt, pBuffer, return_code,
11914             MPTSAS_FW_DIAG_TYPE_RELEASE);
11915         return (status);
```

```
11916 }
11918 static int
11919 mptsas_do_diag_action(mptsas_t *mpt, uint32_t action, uint8_t *diag_action,
11920     uint32_t length, uint32_t *return_code, int ioctl_mode)
11921 {
11922         mptsas_fw_diag_register_t       diag_register;
11923         mptsas_fw_diag_unregister_t     diag_unregister;
11924         mptsas_fw_diag_query_t          diag_query;
11925         mptsas_diag_read_buffer_t       diag_read_buffer;
11926         mptsas_fw_diag_release_t        diag_release;
11927         int                             status = DDI_SUCCESS;
11928         uint32_t                        original_return_code, read_buf_len;

11930         ASSERT(mutex_owned(&mpt->m_mutex));

11932         original_return_code = *return_code;
11933         *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;

11935         switch (action) {
11936                 case MPTSAS_FW_DIAG_TYPE_REGISTER:
11937                         if (!length) {
11938                                 *return_code =
11939                                         MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11940                                 status = DDI_FAILURE;
11941                                 break;
11942                         }
11943                         if (ddi_copyin(diag_action, &diag_register,
11944                             sizeof (diag_register), ioctl_mode) != 0) {
11945                                 return (DDI_FAILURE);
11946                         }
11947                         status = mptsas_diag_register(mpt, &diag_register,
11948                             return_code);
11949                         break;

11951                 case MPTSAS_FW_DIAG_TYPE_UNREGISTER:
11952                         if (length < sizeof (diag_unregister)) {
11953                                 *return_code =
11954                                         MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11955                                 status = DDI_FAILURE;
11956                                 break;
11957                         }
11958                         if (ddi_copyin(diag_action, &diag_unregister,
11959                             sizeof (diag_unregister), ioctl_mode) != 0) {
11960                                 return (DDI_FAILURE);
11961                         }
11962                         status = mptsas_diag_unregister(mpt, &diag_unregister,
11963                             return_code);
11964                         break;

11966                 case MPTSAS_FW_DIAG_TYPE_QUERY:
11967                         if (length < sizeof (diag_query)) {
11968                                 *return_code =
11969                                         MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11970                                 status = DDI_FAILURE;
11971                                 break;
11972                         }
11973                         if (ddi_copyin(diag_action, &diag_query,
11974                             sizeof (diag_query), ioctl_mode) != 0) {
11975                                 return (DDI_FAILURE);
11976                         }
11977                         status = mptsas_diag_query(mpt, &diag_query,
11978                             return_code);
11979                         if (status == DDI_SUCCESS) {
11980                                 if (ddi_copyout(&diag_query, diag_action,
11981                                     sizeof (diag_query), ioctl_mode) != 0) {
```



```
11982                                         return (DDI_FAILURE);
11983                                 }
11984                         }
11985                         break;

11987                 case MPTSAS_FW_DIAG_TYPE_READ_BUFFER:
11988                         if (ddi_copyin(diag_action, &diag_read_buffer,
11989                             sizeof (diag_read_buffer) - 4, ioctl_mode) != 0) {
11990                                 return (DDI_FAILURE);
11991                         }
11992                         read_buf_len = sizeof (diag_read_buffer) -
11993                             sizeof (diag_read_buffer.DataBuffer) +
11994                             diag_read_buffer.BytesToRead;
11995                         if (length < read_buf_len) {
11996                                 *return_code =
11997                                     MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11998                                 status = DDI_FAILURE;
11999                                 break;
12000                         }
12001                         status = mptsas_diag_read_buffer(mpt,
12002                             &diag_read_buffer, diag_action +
12003                             sizeof (diag_read_buffer) - 4, return_code,
12004                             ioctl_mode);
12005                         if (status == DDI_SUCCESS) {
12006                                 if (ddi_copyout(&diag_read_buffer, diag_action,
12007                                     sizeof (diag_read_buffer) - 4, ioctl_mode)
12008                                     != 0) {
12009                                         return (DDI_FAILURE);
12010                                 }
12011                         }
12012                         break;

12014                 case MPTSAS_FW_DIAG_TYPE_RELEASE:
12015                         if (length < sizeof (diag_release)) {
12016                                 *return_code =
12017                                     MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
12018                                 status = DDI_FAILURE;
12019                                 break;
12020                         }
12021                         if (ddi_copyin(diag_action, &diag_release,
12022                             sizeof (diag_release), ioctl_mode) != 0) {
12023                                 return (DDI_FAILURE);
12024                         }
12025                         status = mptsas_diag_release(mpt, &diag_release,
12026                             return_code);
12027                         break;

12029                 default:
12030                         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
12031                         status = DDI_FAILURE;
12032                         break;
12033         }

12035         if ((status == DDI_FAILURE) &&
12036             (original_return_code == MPTSAS_FW_DIAG_NEW) &&
12037             (*return_code != MPTSAS_FW_DIAG_ERROR_SUCCESS)) {
12038                 status = DDI_SUCCESS;
12039         }

12041         return (status);
12042 }

12044 static int
12045 mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *user_data, int mode)
12046 {
12047         int                     status;
```

```
12048         mptsas_diag_action_t    driver_data;

12050         ASSERT(mutex_owned(&mpt->m_mutex));

12052         /*
12053          * Copy the user data to a driver data buffer.
12054          */
12055         if (ddi_copyin(user_data, &driver_data, sizeof (mptsas_diag_action_t),
12056             mode) == 0) {
12057                 /*
12058                  * Send diag action request if Action is valid
12059                  */
12060                 if (driver_data.Action == MPTSAS_FW_DIAG_TYPE_REGISTER ||
12061                     driver_data.Action == MPTSAS_FW_DIAG_TYPE_UNREGISTER ||
12062                     driver_data.Action == MPTSAS_FW_DIAG_TYPE_QUERY ||
12063                     driver_data.Action == MPTSAS_FW_DIAG_TYPE_READ_BUFFER ||
12064                     driver_data.Action == MPTSAS_FW_DIAG_TYPE_RELEASE) {
12065                         status = mptsas_do_diag_action(mpt, driver_data.Action,
12066                             (void *)(uintptr_t)driver_data.PtrDiagAction,
12067                             driver_data.Length, &driver_data.ReturnCode,
12068                             mode);
12069                         if (status == DDI_SUCCESS) {
12070                                 if (ddi_copyout(&driver_data.ReturnCode,
12071                                     &user_data->ReturnCode,
12072                                     sizeof (user_data->ReturnCode), mode)
12073                                     != 0) {
12074                                         status = EFAULT;
12075                                 } else {
12076                                         status = 0;
12077                                 }
12078                         } else {
12079                                 status = EIO;
12080                         }
12081                 } else {
12082                         status = EINVAL;
12083                 }
12084         } else {
12085                 status = EFAULT;
12086         }

12088         return (status);
12089 }

12091 /*
12092  * This routine handles the "event query" ioctl.
12093  */
12094 static int
12095 mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data, int mode,
12096     int *rval)
12097 {
12098         int                     status;
12099         mptsas_event_query_t    driverdata;
12100         uint8_t                 i;

12102         driverdata.Entries = MPTSAS_EVENT_QUEUE_SIZE;

12104         mutex_enter(&mpt->m_mutex);
12105         for (i = 0; i < 4; i++) {
12106                 driverdata.Types[i] = mpt->m_event_mask[i];
12107         }
12108         mutex_exit(&mpt->m_mutex);

12110         if (ddi_copyout(&driverdata, data, sizeof (driverdata), mode) != 0) {
12111                 status = EFAULT;
12112         } else {
12113                 *rval = MPTIOCTL_STATUS_GOOD;
```



```
12114                 status = 0;
12115         }

12117         return (status);
12118 }

12120 /*
12121  * This routine handles the "event enable" ioctl.
12122  */
12123 static int
12124 mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data, int mode,
12125     int *rval)
12126 {
12127         int                     status;
12128         mptsas_event_enable_t   driverdata;
12129         uint8_t                 i;

12131         if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
12132                 mutex_enter(&mpt->m_mutex);
12133                 for (i = 0; i < 4; i++) {
12134                         mpt->m_event_mask[i] = driverdata.Types[i];
12135                 }
12136                 mutex_exit(&mpt->m_mutex);

12138                 *rval = MPTIOCTL_STATUS_GOOD;
12139                 status = 0;
12140         } else {
12141                 status = EFAULT;
12142         }
12143         return (status);
12144 }

12146 /*
12147  * This routine handles the "event report" ioctl.
12148  */
12149 static int
12150 mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data, int mode,
12151     int *rval)
12152 {
12153         int                     status;
12154         mptsas_event_report_t   driverdata;

12156         mutex_enter(&mpt->m_mutex);

12158         if (ddi_copyin(&data->Size, &driverdata.Size, sizeof (driverdata.Size),
12159             mode) == 0) {
12160                 if (driverdata.Size >= sizeof (mpt->m_events)) {
12161                         if (ddi_copyout(mpt->m_events, data->Events,
12162                             sizeof (mpt->m_events), mode) != 0) {
12163                                 status = EFAULT;
12164                         } else {
12165                                 if (driverdata.Size > sizeof (mpt->m_events)) {
12166                                         driverdata.Size =
12167                                             sizeof (mpt->m_events);
12168                                         if (ddi_copyout(&driverdata.Size,
12169                                             &data->Size,
12170                                             sizeof (driverdata.Size),
12171                                             mode) != 0) {
12172                                                 status = EFAULT;
12173                                         } else {
12174                                                 *rval = MPTIOCTL_STATUS_GOOD;
12175                                                 status = 0;
12176                                         }
12177                                 } else {
12178                                         *rval = MPTIOCTL_STATUS_GOOD;
12179                                         status = 0;
```

```
12180                                   }
12181                           }
12182                   } else {
12183                           *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12184                           status = 0;
12185                   }
12186           } else {
12187                   status = EFAULT;
12188           }

12190           mutex_exit(&mpt->m_mutex);
12191           return (status);
12192   }

12194   static void
12195   mptsas_lookup_pci_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12196   {
12197           int     *reg_data;
12198           uint_t  reglen;

12200           /*
12201            * Lookup the 'reg' property and extract the other data
12202            */
12203           if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12204               DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
12205               DDI_PROP_SUCCESS) {
12206                   /*
12207                    * Extract the PCI data from the 'reg' property first DWORD.
12208                    * The entry looks like the following:
12209                    * First DWORD:
12210                    * Bits 0 - 7 8-bit Register number
12211                    * Bits 8 - 10 3-bit Function number
12212                    * Bits 11 - 15 5-bit Device number
12213                    * Bits 16 - 23 8-bit Bus number
12214                    * Bits 24 - 25 2-bit Address Space type identifier
12215                    *
12216                    */
12217                   adapter_data->PciInformation.u.bits.BusNumber =
12218                       (reg_data[0] & 0x00FF0000) >> 16;
12219                   adapter_data->PciInformation.u.bits.DeviceNumber =
12220                       (reg_data[0] & 0x0000F800) >> 11;
12221                   adapter_data->PciInformation.u.bits.FunctionNumber =
12222                       (reg_data[0] & 0x00000700) >> 8;
12223                   ddi_prop_free((void *)reg_data);
12224           } else {
12225                   /*
12226                    * If we can't determine the PCI data then we fill in FF's for
12227                    * the data to indicate this.
12228                    */
12229                   adapter_data->PCIDeviceHwId = 0xFFFFFFFF;
12230                   adapter_data->MpiPortNumber = 0xFFFFFFFF;
12231                   adapter_data->PciInformation.u.AsDWORD = 0xFFFFFFFF;
12232           }

12234           /*
12235            * Saved in the mpt->m_fwversion
12236            */
12237           adapter_data->MpiFirmwareVersion = mpt->m_fwversion;
12238   }

12240   static void
12241   mptsas_read_adapter_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12242   {
12243           char    *driver_verstr = MPTSAS_MOD_STRING;

12245           mptsas_lookup_pci_data(mpt, adapter_data);
```



```
12246           adapter_data->AdapterType = mpt->m_MPI25 ?
12247               MPTIOCTL_ADAPTER_TYPE_SAS3 :
12248               MPTIOCTL_ADAPTER_TYPE_SAS2;
12249           adapter_data->PCIDeviceHwId = (uint32_t)mpt->m_devid;
12250           adapter_data->PCIDeviceHwRev = (uint32_t)mpt->m_revid;
12251           adapter_data->SubSystemId = (uint32_t)mpt->m_ssid;
12252           adapter_data->SubsystemVendorId = (uint32_t)mpt->m_svid;
12253           (void) strcpy((char *)&adapter_data->DriverVersion[0], driver_verstr);
12254           adapter_data->BiosVersion = 0;
12255           (void) mptsas_get_bios_page3(mpt, &adapter_data->BiosVersion);
12256   }

12258   static void
12259   mptsas_read_pci_info(mptsas_t *mpt, mptsas_pci_info_t *pci_info)
12260   {
12261           int     *reg_data, i;
12262           uint_t  reglen;

12264           /*
12265            * Lookup the 'reg' property and extract the other data
12266            */
12267           if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12268               DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
12269               DDI_PROP_SUCCESS) {
12270                   /*
12271                    * Extract the PCI data from the 'reg' property first DWORD.
12272                    * The entry looks like the following:
12273                    * First DWORD:
12274                    * Bits 8 - 10 3-bit Function number
12275                    * Bits 11 - 15 5-bit Device number
12276                    * Bits 16 - 23 8-bit Bus number
12277                    */
12278                   pci_info->BusNumber = (reg_data[0] & 0x00FF0000) >> 16;
12279                   pci_info->DeviceNumber = (reg_data[0] & 0x0000F800) >> 11;
12280                   pci_info->FunctionNumber = (reg_data[0] & 0x00000700) >> 8;
12281                   ddi_prop_free((void *)reg_data);
12282           } else {
12283                   /*
12284                    * If we can't determine the PCI info then we fill in FF's for
12285                    * the data to indicate this.
12286                    */
12287                   pci_info->BusNumber = 0xFFFFFFFF;
12288                   pci_info->DeviceNumber = 0xFF;
12289                   pci_info->FunctionNumber = 0xFF;
12290           }

12292           /*
12293            * Now get the interrupt vector and the pci header.  The vector can
12294            * only be 0 right now.  The header is the first 256 bytes of config
12295            * space.
12296            */
12297           pci_info->InterruptVector = 0;
12298           for (i = 0; i < sizeof (pci_info->PciHeader); i++) {
12299                   pci_info->PciHeader[i] = pci_config_get8(mpt->m_config_handle,
12300                       i);
12301           }
12302   }

12304   static int
12305   mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data, int mode)
12306   {
12307           int                     status = 0;
12308           mptsas_reg_access_t     driverdata;

12310           mutex_enter(&mpt->m_mutex);
12311           if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
```

```
12312                 switch (driverdata.Command) {
12313                         /*
12314                          * IO access is not supported.
12315                          */
12316                         case REG_IO_READ:
12317                         case REG_IO_WRITE:
12318                                 mptsas_log(mpt, CE_WARN, "IO access is not "
12319                                     "supported.  Use memory access.");
12320                                 status = EINVAL;
12321                                 break;

12323                         case REG_MEM_READ:
12324                                 driverdata.RegData = ddi_get32(mpt->m_datap,
12325                                     (uint32_t *)(void *)mpt->m_reg +
12326                                     driverdata.RegOffset);
12327                                 if (ddi_copyout(&driverdata.RegData,
12328                                     &data->RegData,
12329                                     sizeof (driverdata.RegData), mode) != 0) {
12330                                         mptsas_log(mpt, CE_WARN, "Register "
12331                                             "Read Failed");
12332                                         status = EFAULT;
12333                                 }
12334                                 break;

12336                         case REG_MEM_WRITE:
12337                                 ddi_put32(mpt->m_datap,
12338                                     (uint32_t *)(void *)mpt->m_reg +
12339                                     driverdata.RegOffset,
12340                                     driverdata.RegData);
12341                                 break;

12343                         default:
12344                                 status = EINVAL;
12345                                 break;
12346                 }
12347         } else {
12348                 status = EFAULT;
12349         }

12351         mutex_exit(&mpt->m_mutex);
12352         return (status);
12353 }

12355 static int
12356 led_control(mptsas_t *mpt, intptr_t data, int mode)
12357 {
12358         int ret = 0;
12359         mptsas_led_control_t lc;
12360         mptsas_target_t *ptgt;

12362         if (ddi_copyin((void *)data, &lc, sizeof (lc), mode) != 0) {
12363                 return (EFAULT);
12364         }

12366         if ((lc.Command != MPTSAS_LEDCTL_FLAG_SET &&
12367             lc.Command != MPTSAS_LEDCTL_FLAG_GET) ||
12368             lc.Led < MPTSAS_LEDCTL_LED_MIN ||
12369             lc.Led > MPTSAS_LEDCTL_LED_MAX ||
12370             (lc.Command == MPTSAS_LEDCTL_FLAG_SET && lc.LedStatus != 0 &&
12371             lc.LedStatus != 1)) {
12372                 return (EINVAL);
12373         }

12375         if ((lc.Command == MPTSAS_LEDCTL_FLAG_SET && (mode & FWRITE) == 0) ||
12376             (lc.Command == MPTSAS_LEDCTL_FLAG_GET && (mode & FREAD) == 0))
12377                 return (EACCES);
```



```
12379         /* Locate the target we're interrogating... */
12380         mutex_enter(&mpt->m_mutex);
12381         ptgt = refhash_linear_search(mpt->m_targets,
12382             mptsas_target_eval_slot, &lc);
12383         if (ptgt == NULL) {
12384                 /* We could not find a target for that enclosure/slot. */
12385                 mutex_exit(&mpt->m_mutex);
12386                 return (ENOENT);
12387         }

12389         if (lc.Command == MPTSAS_LEDCTL_FLAG_SET) {
12390                 /* Update our internal LED state. */
12391                 ptgt->m_led_status &= ~(1 << (lc.Led - 1));
12392                 ptgt->m_led_status |= lc.LedStatus << (lc.Led - 1);

12394                 /* Flush it to the controller. */
12395                 ret = mptsas_flush_led_status(mpt, ptgt);
12396                 mutex_exit(&mpt->m_mutex);
12397                 return (ret);
12398         }

12400         /* Return our internal LED state. */
12401         lc.LedStatus = (ptgt->m_led_status >> (lc.Led - 1)) & 1;
12402         mutex_exit(&mpt->m_mutex);

12404         if (ddi_copyout(&lc, (void *)data, sizeof (lc), mode) != 0) {
12405                 return (EFAULT);
12406         }

12408         return (0);
12409 }

12411 static int
12412 get_disk_info(mptsas_t *mpt, intptr_t data, int mode)
12413 {
12414         uint16_t i = 0;
12415         uint16_t count = 0;
12416         int ret = 0;
12417         mptsas_target_t *ptgt;
12418         mptsas_disk_info_t *di;
12419         STRUCT_DECL(mptsas_get_disk_info, gdi);

12421         if ((mode & FREAD) == 0)
12422                 return (EACCES);

12424         STRUCT_INIT(gdi, get_udatamodel());

12426         if (ddi_copyin((void *)data, STRUCT_BUF(gdi), STRUCT_SIZE(gdi),
12427             mode) != 0) {
12428                 return (EFAULT);
12429         }

12431         /* Find out how many targets there are. */
12432         mutex_enter(&mpt->m_mutex);
12433         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12434             ptgt = refhash_next(mpt->m_targets, ptgt)) {
12435                 count++;
12436         }
12437         mutex_exit(&mpt->m_mutex);

12439         /*
12440          * If we haven't been asked to copy out information on each target,
12441          * then just return the count.
12442          */
12443         STRUCT_FSET(gdi, DiskCount, count);
```

```
12444         if (STRUCT_FGETP(gdi, PtrDiskInfoArray) == NULL)
12445                 goto copy_out;

12447         /*
12448          * If we haven't been given a large enough buffer to copy out into,
12449          * let the caller know.
12450          */
12451         if (STRUCT_FGET(gdi, DiskInfoArraySize) <
12452             count * sizeof (mptsas_disk_info_t)) {
12453                 ret = ENOSPC;
12454                 goto copy_out;
12455         }

12457         di = kmem_zalloc(count * sizeof (mptsas_disk_info_t), KM_SLEEP);

12459         mutex_enter(&mpt->m_mutex);
12460         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12461             ptgt = refhash_next(mpt->m_targets, ptgt)) {
12462                 if (i >= count) {
12463                         /*
12464                          * The number of targets changed while we weren't
12465                          * looking, so give up.
12466                          */
12467                         refhash_rele(mpt->m_targets, ptgt);
12468                         mutex_exit(&mpt->m_mutex);
12469                         kmem_free(di, count * sizeof (mptsas_disk_info_t));
12470                         return (EAGAIN);
12471                 }
12472                 di[i].Instance = mpt->m_instance;
12473                 di[i].Enclosure = ptgt->m_enclosure;
12474                 di[i].Slot = ptgt->m_slot_num;
12475                 di[i].SasAddress = ptgt->m_addr.mta_wwn;
12476                 i++;
12477         }
12478         mutex_exit(&mpt->m_mutex);
12479         STRUCT_FSET(gdi, DiskCount, i);

12481         /* Copy out the disk information to the caller. */
12482         if (ddi_copyout((void *)di, STRUCT_FGETP(gdi, PtrDiskInfoArray),
12483             i * sizeof (mptsas_disk_info_t), mode) != 0) {
12484                 ret = EFAULT;
12485         }

12487         kmem_free(di, count * sizeof (mptsas_disk_info_t));

12489 copy_out:
12490         if (ddi_copyout(STRUCT_BUF(gdi), (void *)data, STRUCT_SIZE(gdi),
12491             mode) != 0) {
12492                 ret = EFAULT;
12493         }

12495         return (ret);
12496 }

12498 static int
12499 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
12500     int *rval)
12501 {
12502         int                     status = 0;
12503         mptsas_t                *mpt;
12504         mptsas_update_flash_t   flashdata;
12505         mptsas_pass_thru_t      passthru_data;
12506         mptsas_adapter_data_t   adapter_data;
12507         mptsas_pci_info_t       pci_info;
12508         int                     copylen;
```



```
12510         int                     iport_flag = 0;
12511         dev_info_t              *dip = NULL;
12512         mptsas_phymask_t        phymask = 0;
12513         struct devctl_iocdata   *dcp = NULL;
12514         char                    *addr = NULL;
12515         mptsas_target_t         *ptgt = NULL;

12517         *rval = MPTIOCTL_STATUS_GOOD;
12518         if (secpolicy_sys_config(credp, B_FALSE) != 0) {
12519                 return (EPERM);
12520         }

12522         mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
12523         if (mpt == NULL) {
12524                 /*
12525                  * Called from iport node, get the states
12526                  */
12527                 iport_flag = 1;
12528                 dip = mptsas_get_dip_from_dev(dev, &phymask);
12529                 if (dip == NULL) {
12530                         return (ENXIO);
12531                 }
12532                 mpt = DIP2MPT(dip);
12533         }
12534         /* Make sure power level is D0 before accessing registers */
12535         mutex_enter(&mpt->m_mutex);
12536         if (mpt->m_options & MPTSAS_OPT_PM) {
12537                 (void) pm_busy_component(mpt->m_dip, 0);
12538                 if (mpt->m_power_level != PM_LEVEL_D0) {
12539                         mutex_exit(&mpt->m_mutex);
12540                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
12541                             DDI_SUCCESS) {
12542                                 mptsas_log(mpt, CE_WARN,
12543                                     "mptsas%d: mptsas_ioctl: Raise power "
12544                                     "request failed.", mpt->m_instance);
12545                                 (void) pm_idle_component(mpt->m_dip, 0);
12546                                 return (ENXIO);
12547                         }
12548                 } else {
12549                         mutex_exit(&mpt->m_mutex);
12550                 }
12551         } else {
12552                 mutex_exit(&mpt->m_mutex);
12553         }

12555         if (iport_flag) {
12556                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
12557                 if (status != 0) {
12558                         goto out;
12559                 }
12560                 /*
12561                  * The following code control the OK2RM LED, it doesn't affect
12562                  * the ioctl return status.
12563                  */
12564                 if ((cmd == DEVCTL_DEVICE_ONLINE) ||
12565                     (cmd == DEVCTL_DEVICE_OFFLINE)) {
12566                         if (ndi_dc_allochdl((void *)data, &dcp) !=
12567                             NDI_SUCCESS) {
12568                                 goto out;
12569                         }
12570                         addr = ndi_dc_getaddr(dcp);
12571                         ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
12572                         if (ptgt == NULL) {
12573                                 NDBG14(("mptsas_ioctl led control: tgt %s not "
12574                                     "found", addr));
12575                                 ndi_dc_freehdl(dcp);
```

```
12576                                   goto out;
12577                           }
12578                           mutex_enter(&mpt->m_mutex);
12579                           if (cmd == DEVCTL_DEVICE_ONLINE) {
12580                                   ptgt->m_tgt_unconfigured = 0;
12581                           } else if (cmd == DEVCTL_DEVICE_OFFLINE) {
12582                                   ptgt->m_tgt_unconfigured = 1;
12583                           }
12584                           if (cmd == DEVCTL_DEVICE_OFFLINE) {
12585                                   ptgt->m_led_status |=
12586                                       (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12587                           } else {
12588                                   ptgt->m_led_status &=
12589                                       ~(1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12590                           }
12591                           (void) mptsas_flush_led_status(mpt, ptgt);
12592                           mutex_exit(&mpt->m_mutex);
12593                           ndi_dc_freehdl(dcp);
12594                   }
12595                   goto out;
12596           }
12597           switch (cmd) {
12598                   case MPTIOCTL_GET_DISK_INFO:
12599                           status = get_disk_info(mpt, data, mode);
12600                           break;
12601                   case MPTIOCTL_LED_CONTROL:
12602                           status = led_control(mpt, data, mode);
12603                           break;
12604                   case MPTIOCTL_UPDATE_FLASH:
12605                           if (ddi_copyin((void *)data, &flashdata,
12606                                   sizeof (struct mptsas_update_flash), mode)) {
12607                                   status = EFAULT;
12608                                   break;
12609                           }

12611                           mutex_enter(&mpt->m_mutex);
12612                           if (mptsas_update_flash(mpt,
12613                               (caddr_t)(long)flashdata.PtrBuffer,
12614                               flashdata.ImageSize, flashdata.ImageType, mode)) {
12615                                   status = EFAULT;
12616                           }

12618                           /*
12619                            * Reset the chip to start using the new
12620                            * firmware.  Reset if failed also.
12621                            */
12622                           mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12623                           if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
12624                                   status = EFAULT;
12625                           }
12626                           mutex_exit(&mpt->m_mutex);
12627                           break;
12628                   case MPTIOCTL_PASS_THRU:
12629                           /*
12630                            * The user has requested to pass through a command to
12631                            * be executed by the MPT firmware.  Call our routine
12632                            * which does this.  Only allow one passthru IOCTL at
12633                            * one time. Other threads will block on
12634                            * m_passthru_mutex, which is of adaptive variant.
12635                            */
12636                           if (ddi_copyin((void *)data, &passthru_data,
12637                               sizeof (mptsas_pass_thru_t), mode)) {
12638                                   status = EFAULT;
12639                                   break;
12640                           }
12641                           mutex_enter(&mpt->m_passthru_mutex);
```



```
12642                           mutex_enter(&mpt->m_mutex);
12643                           status = mptsas_pass_thru(mpt, &passthru_data, mode);
12644                           mutex_exit(&mpt->m_mutex);
12645                           mutex_exit(&mpt->m_passthru_mutex);

12647                           break;
12648                   case MPTIOCTL_GET_ADAPTER_DATA:
12649                           /*
12650                            * The user has requested to read adapter data.  Call
12651                            * our routine which does this.
12652                            */
12653                           bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
12654                           if (ddi_copyin((void *)data, (void *)&adapter_data,
12655                               sizeof (mptsas_adapter_data_t), mode)) {
12656                                   status = EFAULT;
12657                                   break;
12658                           }
12659                           if (adapter_data.StructureLength >=
12660                               sizeof (mptsas_adapter_data_t)) {
12661                                   adapter_data.StructureLength = (uint32_t)
12662                                       sizeof (mptsas_adapter_data_t);
12663                                   copylen = sizeof (mptsas_adapter_data_t);
12664                                   mutex_enter(&mpt->m_mutex);
12665                                   mptsas_read_adapter_data(mpt, &adapter_data);
12666                                   mutex_exit(&mpt->m_mutex);
12667                           } else {
12668                                   adapter_data.StructureLength = (uint32_t)
12669                                       sizeof (mptsas_adapter_data_t);
12670                                   copylen = sizeof (adapter_data.StructureLength);
12671                                   *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12672                           }
12673                           if (ddi_copyout((void *)(&adapter_data), (void *)data,
12674                               copylen, mode) != 0) {
12675                                   status = EFAULT;
12676                           }
12677                           break;
12678                   case MPTIOCTL_GET_PCI_INFO:
12679                           /*
12680                            * The user has requested to read pci info.  Call
12681                            * our routine which does this.
12682                            */
12683                           bzero(&pci_info, sizeof (mptsas_pci_info_t));
12684                           mutex_enter(&mpt->m_mutex);
12685                           mptsas_read_pci_info(mpt, &pci_info);
12686                           mutex_exit(&mpt->m_mutex);
12687                           if (ddi_copyout((void *)(&pci_info), (void *)data,
12688                               sizeof (mptsas_pci_info_t), mode) != 0) {
12689                                   status = EFAULT;
12690                           }
12691                           break;
12692                   case MPTIOCTL_RESET_ADAPTER:
12693                           mutex_enter(&mpt->m_mutex);
12694                           mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12695                           if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
12696                                   mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
12697                                       "failed");
12698                                   status = EFAULT;
12699                           }
12700                           mutex_exit(&mpt->m_mutex);
12701                           break;
12702                   case MPTIOCTL_DIAG_ACTION:
12703                           /*
12704                            * The user has done a diag buffer action.  Call our
12705                            * routine which does this.  Only allow one diag action
12706                            * at one time.
12707                            */
```

```
12708                           mutex_enter(&mpt->m_mutex);
12709                           if (mpt->m_diag_action_in_progress) {
12710                                   mutex_exit(&mpt->m_mutex);
12711                                   return (EBUSY);
12712                           }
12713                           mpt->m_diag_action_in_progress = 1;
12714                           status = mptsas_diag_action(mpt,
12715                               (mptsas_diag_action_t *)data, mode);
12716                           mpt->m_diag_action_in_progress = 0;
12717                           mutex_exit(&mpt->m_mutex);
12718                           break;
12719                   case MPTIOCTL_EVENT_QUERY:
12720                           /*
12721                            * The user has done an event query. Call our routine
12722                            * which does this.
12723                            */
12724                           status = mptsas_event_query(mpt,
12725                               (mptsas_event_query_t *)data, mode, rval);
12726                           break;
12727                   case MPTIOCTL_EVENT_ENABLE:
12728                           /*
12729                            * The user has done an event enable. Call our routine
12730                            * which does this.
12731                            */
12732                           status = mptsas_event_enable(mpt,
12733                               (mptsas_event_enable_t *)data, mode, rval);
12734                           break;
12735                   case MPTIOCTL_EVENT_REPORT:
12736                           /*
12737                            * The user has done an event report. Call our routine
12738                            * which does this.
12739                            */
12740                           status = mptsas_event_report(mpt,
12741                               (mptsas_event_report_t *)data, mode, rval);
12742                           break;
12743                   case MPTIOCTL_REG_ACCESS:
12744                           /*
12745                            * The user has requested register access.  Call our
12746                            * routine which does this.
12747                            */
12748                           status = mptsas_reg_access(mpt,
12749                               (mptsas_reg_access_t *)data, mode);
12750                           break;
12751                   default:
12752                           status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
12753                               rval);
12754                           break;
12755           }

12757 out:
12758           return (status);
12759 }

12761 int
12762 mptsas_restart_ioc(mptsas_t *mpt)
12763 {
12764           int             rval = DDI_SUCCESS;
12765           mptsas_target_t *ptgt = NULL;

12767           ASSERT(mutex_owned(&mpt->m_mutex));

12769           /*
12770            * Set a flag telling I/O path that we're processing a reset.  This is
12771            * needed because after the reset is complete, the hash table still
12772            * needs to be rebuilt.  If I/Os are started before the hash table is
12773            * rebuilt, I/O errors will occur.  This flag allows I/Os to be marked
```



```
12774            * so that they can be retried.
12775            */
12776           mpt->m_in_reset = TRUE;

12778           /*
12779            * Set all throttles to HOLD
12780            */
12781           for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12782               ptgt = refhash_next(mpt->m_targets, ptgt)) {
12783                   mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
12784           }

12786           /*
12787            * Disable interrupts
12788            */
12789           MPTSAS_DISABLE_INTR(mpt);

12791           /*
12792            * Abort all commands: outstanding commands, commands in waitq and
12793            * tx_waitq.
12794            */
12795           mptsas_flush_hba(mpt);

12797           /*
12798            * Reinitialize the chip.
12799            */
12800           if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
12801                   rval = DDI_FAILURE;
12802           }

12804           /*
12805            * Enable interrupts again
12806            */
12807           MPTSAS_ENABLE_INTR(mpt);

12809           /*
12810            * If mptsas_init_chip was successful, update the driver data.
12811            */
12812           if (rval == DDI_SUCCESS) {
12813                   mptsas_update_driver_data(mpt);
12814           }

12816           /*
12817            * Reset the throttles
12818            */
12819           for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12820               ptgt = refhash_next(mpt->m_targets, ptgt)) {
12821                   mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
12822           }

12824           mptsas_doneq_empty(mpt);
12825           mptsas_restart_hba(mpt);

12827           if (rval != DDI_SUCCESS) {
12828                   mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
12829                   ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
12830           }

12832           /*
12833            * Clear the reset flag so that I/Os can continue.
12834            */
12835           mpt->m_in_reset = FALSE;

12837           return (rval);
12838 }
```

```
12840 static int
12841 mptsas_init_chip(mptsas_t *mpt, int first_time)
12842 {
12843         ddi_dma_cookie_t        cookie;
12844         uint32_t                i;
12845         int                     rval;

12847         /*
12848          * Check to see if the firmware image is valid
12849          */
12850         if (ddi_get32(mpt->m_datap, &mpt->m_reg->HostDiagnostic) &
12851             MPI2_DIAG_FLASH_BAD_SIG) {
12852                 mptsas_log(mpt, CE_WARN, "mptsas bad flash signature!");
12853                 goto fail;
12854         }

12856         /*
12857          * Reset the chip
12858          */
12859         rval = mptsas_ioc_reset(mpt, first_time);
12860         if (rval == MPTSAS_RESET_FAIL) {
12861                 mptsas_log(mpt, CE_WARN, "hard reset failed!");
12862                 goto fail;
12863         }

12865         if ((rval == MPTSAS_SUCCESS_MUR) && (!first_time)) {
12866                 goto mur;
12867         }
12868         /*
12869          * Setup configuration space
12870          */
12871         if (mptsas_config_space_init(mpt) == FALSE) {
12872                 mptsas_log(mpt, CE_WARN, "mptsas_config_space_init "
12873                     "failed!");
12874                 goto fail;
12875         }

12877         /*
12878          * IOC facts can change after a diag reset so all buffers that are
12879          * based on these numbers must be de-allocated and re-allocated.  Get
12880          * new IOC facts each time chip is initialized.
12881          */
12882         if (mptsas_ioc_get_facts(mpt) == DDI_FAILURE) {
12883                 mptsas_log(mpt, CE_WARN, "mptsas_ioc_get_facts failed");
12884                 goto fail;
12885         }

12887         if (mptsas_alloc_active_slots(mpt, KM_SLEEP)) {
12888                 goto fail;
12889         }
12890         /*
12891          * Allocate request message frames, reply free queue, reply descriptor
12892          * post queue, and reply message frames using latest IOC facts.
12893          */
12894         if (mptsas_alloc_request_frames(mpt) == DDI_FAILURE) {
12895                 mptsas_log(mpt, CE_WARN, "mptsas_alloc_request_frames failed");
12896                 goto fail;
12897         }
12898         if (mptsas_alloc_sense_bufs(mpt) == DDI_FAILURE) {
12899                 mptsas_log(mpt, CE_WARN, "mptsas_alloc_sense_bufs failed");
12900                 goto fail;
12901         }
12902         if (mptsas_alloc_free_queue(mpt) == DDI_FAILURE) {
12903                 mptsas_log(mpt, CE_WARN, "mptsas_alloc_free_queue failed!");
12904                 goto fail;
12905         }
```



```
12906         if (mptsas_alloc_post_queue(mpt) == DDI_FAILURE) {
12907                 mptsas_log(mpt, CE_WARN, "mptsas_alloc_post_queue failed!");
12908                 goto fail;
12909         }
12910         if (mptsas_alloc_reply_frames(mpt) == DDI_FAILURE) {
12911                 mptsas_log(mpt, CE_WARN, "mptsas_alloc_reply_frames failed!");
12912                 goto fail;
12913         }

12915 mur:
12916         /*
12917          * Re-Initialize ioc to operational state
12918          */
12919         if (mptsas_ioc_init(mpt) == DDI_FAILURE) {
12920                 mptsas_log(mpt, CE_WARN, "mptsas_ioc_init failed");
12921                 goto fail;
12922         }

12924         mptsas_alloc_reply_args(mpt);

12926         /*
12927          * Initialize reply post index.  Reply free index is initialized after
12928          * the next loop.
12929          */
12930         mpt->m_post_index = 0;

12932         /*
12933          * Initialize the Reply Free Queue with the physical addresses of our
12934          * reply frames.
12935          */
12936         cookie.dmac_address = mpt->m_reply_frame_dma_addr & 0xffffffffu;
12937         for (i = 0; i < mpt->m_max_replies; i++) {
12938                 ddi_put32(mpt->m_acc_free_queue_hdl,
12939                     &((uint32_t *)(void *)mpt->m_free_queue)[i],
12940                     cookie.dmac_address);
12941                 cookie.dmac_address += mpt->m_reply_frame_size;
12942         }
12943         (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
12944             DDI_DMA_SYNC_FORDEV);

12946         /*
12947          * Initialize the reply free index to one past the last frame on the
12948          * queue.  This will signify that the queue is empty to start with.
12949          */
12950         mpt->m_free_index = i;
12951         ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex, i);

12953         /*
12954          * Initialize the reply post queue to 0xFFFFFFFF,0xFFFFFFFF's.
12955          */
12956         for (i = 0; i < mpt->m_post_queue_depth; i++) {
12957                 ddi_put64(mpt->m_acc_post_queue_hdl,
12958                     &((uint64_t *)(void *)mpt->m_post_queue)[i],
12959                     0xFFFFFFFFFFFFFFFF);
12960         }
12961         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
12962             DDI_DMA_SYNC_FORDEV);

12964         /*
12965          * Enable ports
12966          */
12967         if (mptsas_ioc_enable_port(mpt) == DDI_FAILURE) {
12968                 mptsas_log(mpt, CE_WARN, "mptsas_ioc_enable_port failed");
12969                 goto fail;
12970         }
```

```
12972         /*
12973          * enable events
12974          */
12975         if (mptsas_ioc_enable_event_notification(mpt)) {
12976                 mptsas_log(mpt, CE_WARN,
12977                     "mptsas_ioc_enable_event_notification failed");
12978                 goto fail;
12979         }

12981         /*
12982          * We need checks in attach and these.
12983          * chip_init is called in mult. places
12984          */

12986         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
12987             DDI_SUCCESS) ||
12988             (mptsas_check_dma_handle(mpt->m_dma_req_sense_hdl) !=
12989             DDI_SUCCESS) ||
12990             (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
12991             DDI_SUCCESS) ||
12992             (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
12993             DDI_SUCCESS) ||
12994             (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
12995             DDI_SUCCESS) ||
12996             (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=
12997             DDI_SUCCESS)) {
12998                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
12999                 goto fail;
13000         }

13002         /* Check all acc handles */
13003         if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
13004             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
13005             DDI_SUCCESS) ||
13006             (mptsas_check_acc_handle(mpt->m_acc_req_sense_hdl) !=
13007             DDI_SUCCESS) ||
13008             (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
13009             DDI_SUCCESS) ||
13010             (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
13011             DDI_SUCCESS) ||
13012             (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
13013             DDI_SUCCESS) ||
13014             (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
13015             DDI_SUCCESS) ||
13016             (mptsas_check_acc_handle(mpt->m_config_handle) !=
13017             DDI_SUCCESS)) {
13018                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
13019                 goto fail;
13020         }

13022         return (DDI_SUCCESS);

13024 fail:
13025         return (DDI_FAILURE);
13026 }

13028 static int
13029 mptsas_get_pci_cap(mptsas_t *mpt)
13030 {
13031         ushort_t caps_ptr, cap, cap_count;

13033         if (mpt->m_config_handle == NULL)
13034                 return (FALSE);
13035         /*
13036          * Check if capabilities list is supported and if so,
13037          * get initial capabilities pointer and clear bits 0,1.
```



```
13038          */
13039         if (pci_config_get16(mpt->m_config_handle, PCI_CONF_STAT)
13040             & PCI_STAT_CAP) {
13041                 caps_ptr = P2ALIGN(pci_config_get8(mpt->m_config_handle,
13042                     PCI_CONF_CAP_PTR), 4);
13043         } else {
13044                 caps_ptr = PCI_CAP_NEXT_PTR_NULL;
13045         }

13047         /*
13048          * Walk capabilities if supported.
13049          */
13050         for (cap_count = 0; caps_ptr != PCI_CAP_NEXT_PTR_NULL; ) {

13052                 /*
13053                  * Check that we haven't exceeded the maximum number of
13054                  * capabilities and that the pointer is in a valid range.
13055                  */
13056                 if (++cap_count > 48) {
13057                         mptsas_log(mpt, CE_WARN,
13058                             "too many device capabilities.\n");
13059                         break;
13060                 }
13061                 if (caps_ptr < 64) {
13062                         mptsas_log(mpt, CE_WARN,
13063                             "capabilities pointer 0x%x out of range.\n",
13064                             caps_ptr);
13065                         break;
13066                 }

13068                 /*
13069                  * Get next capability and check that it is valid.
13070                  * For now, we only support power management.
13071                  */
13072                 cap = pci_config_get8(mpt->m_config_handle, caps_ptr);
13073                 switch (cap) {
13074                         case PCI_CAP_ID_PM:
13075                                 mptsas_log(mpt, CE_NOTE,
13076                                     "?mptsas%d supports power management.\n",
13077                                     mpt->m_instance);
13078                                 mpt->m_options |= MPTSAS_OPT_PM;

13080                                 /* Save PMCSR offset */
13081                                 mpt->m_pmcsr_offset = caps_ptr + PCI_PMCSR;
13082                                 break;
13083                         /*
13084                          * The following capabilities are valid.  Any others
13085                          * will cause a message to be logged.
13086                          */
13087                         case PCI_CAP_ID_VPD:
13088                         case PCI_CAP_ID_MSI:
13089                         case PCI_CAP_ID_PCIX:
13090                         case PCI_CAP_ID_PCI_E:
13091                         case PCI_CAP_ID_MSI_X:
13092                                 break;
13093                         default:
13094                                 mptsas_log(mpt, CE_NOTE,
13095                                     "?mptsas%d unrecognized capability "
13096                                     "0x%x.\n", mpt->m_instance, cap);
13097                                 break;
13098                 }

13100                 /*
13101                  * Get next capabilities pointer and clear bits 0,1.
13102                  */
13103                 caps_ptr = P2ALIGN(pci_config_get8(mpt->m_config_handle,
```

```
13104                     (caps_ptr + PCI_CAP_NEXT_PTR)), 4);
13105         }
13106         return (TRUE);
13107 }

13109 static int
13110 mptsas_init_pm(mptsas_t *mpt)
13111 {
13112         char            pmc_name[16];
13113         char            *pmc[] = {
13114                               NULL,
13115                               "0=Off (PCI D3 State)",
13116                               "3=On (PCI D0 State)",
13117                               NULL
13118                         };
13119         uint16_t        pmcsr_stat;

13121         if (mptsas_get_pci_cap(mpt) == FALSE) {
13122                 return (DDI_FAILURE);
13123         }
13124         /*
13125          * If PCI's capability does not support PM, then don't need
13126          * to registe the pm-components
13127          */
13128         if (!(mpt->m_options & MPTSAS_OPT_PM))
13129                 return (DDI_SUCCESS);
13130         /*
13131          * If power management is supported by this chip, create
13132          * pm-components property for the power management framework
13133          */
13134         (void) sprintf(pmc_name, "NAME=mptsas%d", mpt->m_instance);
13135         pmc[0] = pmc_name;
13136         if (ddi_prop_update_string_array(DDI_DEV_T_NONE, mpt->m_dip,
13137             "pm-components", pmc, 3) != DDI_PROP_SUCCESS) {
13138                 mpt->m_options &= ~MPTSAS_OPT_PM;
13139                 mptsas_log(mpt, CE_WARN,
13140                     "mptsas%d: pm-component property creation failed.",
13141                     mpt->m_instance);
13142                 return (DDI_FAILURE);
13143         }

13145         /*
13146          * Power on device.
13147          */
13148         (void) pm_busy_component(mpt->m_dip, 0);
13149         pmcsr_stat = pci_config_get16(mpt->m_config_handle,
13150             mpt->m_pmcsr_offset);
13151         if ((pmcsr_stat & PCI_PMCSR_STATE_MASK) != PCI_PMCSR_D0) {
13152                 mptsas_log(mpt, CE_WARN, "mptsas%d: Power up the device",
13153                     mpt->m_instance);
13154                 pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset,
13155                     PCI_PMCSR_D0);
13156         }
13157         if (pm_power_has_changed(mpt->m_dip, 0, PM_LEVEL_D0) != DDI_SUCCESS) {
13158                 mptsas_log(mpt, CE_WARN, "pm_power_has_changed failed");
13159                 return (DDI_FAILURE);
13160         }
13161         mpt->m_power_level = PM_LEVEL_D0;
13162         /*
13163          * Set pm idle delay.
13164          */
13165         mpt->m_pm_idle_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
13166             mpt->m_dip, 0, "mptsas-pm-idle-delay", MPTSAS_PM_IDLE_TIMEOUT);

13168         return (DDI_SUCCESS);
13169 }
```



```
13171 static int
13172 mptsas_register_intrs(mptsas_t *mpt)
13173 {
13174         dev_info_t *dip;
13175         int intr_types;

13177         dip = mpt->m_dip;

13179         /* Get supported interrupt types */
13180         if (ddi_intr_get_supported_types(dip, &intr_types) != DDI_SUCCESS) {
13181                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_supported_types "
13182                     "failed\n");
13183                 return (FALSE);
13184         }

13186         NDBG6(("ddi_intr_get_supported_types() returned: 0x%x", intr_types));

13188         /*
13189          * Try MSI, but fall back to FIXED
13190          */
13191         if (mptsas_enable_msi && (intr_types & DDI_INTR_TYPE_MSI)) {
13192                 if (mptsas_add_intrs(mpt, DDI_INTR_TYPE_MSI) == DDI_SUCCESS) {
13193                         NDBG0(("Using MSI interrupt type"));
13194                         mpt->m_intr_type = DDI_INTR_TYPE_MSI;
13195                         return (TRUE);
13196                 }
13197         }
13198         if (intr_types & DDI_INTR_TYPE_FIXED) {
13199                 if (mptsas_add_intrs(mpt, DDI_INTR_TYPE_FIXED) == DDI_SUCCESS) {
13200                         NDBG0(("Using FIXED interrupt type"));
13201                         mpt->m_intr_type = DDI_INTR_TYPE_FIXED;
13202                         return (TRUE);
13203                 } else {
13204                         NDBG0(("FIXED interrupt registration failed"));
13205                         return (FALSE);
13206                 }
13207         }

13209         return (FALSE);
13210 }

13212 static void
13213 mptsas_unregister_intrs(mptsas_t *mpt)
13214 {
13215         mptsas_rem_intrs(mpt);
13216 }

13218 /*
13219  * mptsas_add_intrs:
13220  *
13221  * Register FIXED or MSI interrupts.
13222  */
13223 static int
13224 mptsas_add_intrs(mptsas_t *mpt, int intr_type)
13225 {
13226         dev_info_t      *dip = mpt->m_dip;
13227         int             avail, actual, count = 0;
13228         int             i, flag, ret;

13230         NDBG6(("mptsas_add_intrs:interrupt type 0x%x", intr_type));

13232         /* Get number of interrupts */
13233         ret = ddi_intr_get_nintrs(dip, intr_type, &count);
13234         if ((ret != DDI_SUCCESS) || (count <= 0)) {
13235                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_nintrs() failed, "
```

```
13236                       "ret %d count %d\n", ret, count);

13238                   return (DDI_FAILURE);
13239           }

13241           /* Get number of available interrupts */
13242           ret = ddi_intr_get_navail(dip, intr_type, &avail);
13243           if ((ret != DDI_SUCCESS) || (avail == 0)) {
13244                   mptsas_log(mpt, CE_WARN, "ddi_intr_get_navail() failed, "
13245                       "ret %d avail %d\n", ret, avail);

13247                   return (DDI_FAILURE);
13248           }

13250           if (avail < count) {
13251                   mptsas_log(mpt, CE_NOTE, "ddi_intr_get_nvail returned %d, "
13252                       "navail() returned %d", count, avail);
13253           }

13255           /* Mpt only have one interrupt routine */
13256           if ((intr_type == DDI_INTR_TYPE_MSI) && (count > 1)) {
13257                   count = 1;
13258           }

13260           /* Allocate an array of interrupt handles */
13261           mpt->m_intr_size = count * sizeof (ddi_intr_handle_t);
13262           mpt->m_htable = kmem_alloc(mpt->m_intr_size, KM_SLEEP);

13264           flag = DDI_INTR_ALLOC_NORMAL;

13266           /* call ddi_intr_alloc() */
13267           ret = ddi_intr_alloc(dip, mpt->m_htable, intr_type, 0,
13268               count, &actual, flag);

13270           if ((ret != DDI_SUCCESS) || (actual == 0)) {
13271                   mptsas_log(mpt, CE_WARN, "ddi_intr_alloc() failed, ret %d\n",
13272                       ret);
13273                   kmem_free(mpt->m_htable, mpt->m_intr_size);
13274                   return (DDI_FAILURE);
13275           }

13277           /* use interrupt count returned or abort? */
13278           if (actual < count) {
13279                   mptsas_log(mpt, CE_NOTE, "Requested: %d, Received: %d\n",
13280                       count, actual);
13281           }

13283           mpt->m_intr_cnt = actual;

13285           /*
13286            * Get priority for first msi, assume remaining are all the same
13287            */
13288           if ((ret = ddi_intr_get_pri(mpt->m_htable[0],
13289               &mpt->m_intr_pri)) != DDI_SUCCESS) {
13290                   mptsas_log(mpt, CE_WARN, "ddi_intr_get_pri() failed %d\n", ret);

13292                   /* Free already allocated intr */
13293                   for (i = 0; i < actual; i++) {
13294                           (void) ddi_intr_free(mpt->m_htable[i]);
13295                   }

13297                   kmem_free(mpt->m_htable, mpt->m_intr_size);
13298                   return (DDI_FAILURE);
13299           }

13301           /* Test for high level mutex */
```



```
13302           if (mpt->m_intr_pri >= ddi_intr_get_hilevel_pri()) {
13303                   mptsas_log(mpt, CE_WARN, "mptsas_add_intrs: "
13304                       "Hi level interrupt not supported\n");

13306                   /* Free already allocated intr */
13307                   for (i = 0; i < actual; i++) {
13308                           (void) ddi_intr_free(mpt->m_htable[i]);
13309                   }

13311                   kmem_free(mpt->m_htable, mpt->m_intr_size);
13312                   return (DDI_FAILURE);
13313           }

13315           /* Call ddi_intr_add_handler() */
13316           for (i = 0; i < actual; i++) {
13317                   if ((ret = ddi_intr_add_handler(mpt->m_htable[i], mptsas_intr,
13318                       (caddr_t)mpt, (caddr_t)(uintptr_t)i)) != DDI_SUCCESS) {
13319                           mptsas_log(mpt, CE_WARN, "ddi_intr_add_handler() "
13320                               "failed %d\n", ret);

13322                           /* Free already allocated intr */
13323                           for (i = 0; i < actual; i++) {
13324                                   (void) ddi_intr_free(mpt->m_htable[i]);
13325                           }

13327                           kmem_free(mpt->m_htable, mpt->m_intr_size);
13328                           return (DDI_FAILURE);
13329                   }
13330           }

13332           if ((ret = ddi_intr_get_cap(mpt->m_htable[0], &mpt->m_intr_cap))
13333               != DDI_SUCCESS) {
13334                   mptsas_log(mpt, CE_WARN, "ddi_intr_get_cap() failed %d\n", ret);

13336                   /* Free already allocated intr */
13337                   for (i = 0; i < actual; i++) {
13338                           (void) ddi_intr_free(mpt->m_htable[i]);
13339                   }

13341                   kmem_free(mpt->m_htable, mpt->m_intr_size);
13342                   return (DDI_FAILURE);
13343           }

13345           /*
13346            * Enable interrupts
13347            */
13348           if (mpt->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
13349                   /* Call ddi_intr_block_enable() for MSI interrupts */
13350                   (void) ddi_intr_block_enable(mpt->m_htable, mpt->m_intr_cnt);
13351           } else {
13352                   /* Call ddi_intr_enable for MSI or FIXED interrupts */
13353                   for (i = 0; i < mpt->m_intr_cnt; i++) {
13354                           (void) ddi_intr_enable(mpt->m_htable[i]);
13355                   }
13356           }
13357           return (DDI_SUCCESS);
13358   }

13360   /*
13361    * mptsas_rem_intrs:
13362    *
13363    * Unregister FIXED or MSI interrupts
13364    */
13365   static void
13366   mptsas_rem_intrs(mptsas_t *mpt)
13367   {
```

```
13368         int     i;

13370         NDBG6(("mptsas_rem_intrs"));

13372         /* Disable all interrupts */
13373         if (mpt->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
13374                 /* Call ddi_intr_block_disable() */
13375                 (void) ddi_intr_block_disable(mpt->m_htable, mpt->m_intr_cnt);
13376         } else {
13377                 for (i = 0; i < mpt->m_intr_cnt; i++) {
13378                         (void) ddi_intr_disable(mpt->m_htable[i]);
13379                 }
13380         }

13382         /* Call ddi_intr_remove_handler() */
13383         for (i = 0; i < mpt->m_intr_cnt; i++) {
13384                 (void) ddi_intr_remove_handler(mpt->m_htable[i]);
13385                 (void) ddi_intr_free(mpt->m_htable[i]);
13386         }

13388         kmem_free(mpt->m_htable, mpt->m_intr_size);
13389 }

13391 /*
13392  * The IO fault service error handling callback function
13393  */
13394 /*ARGSUSED*/
13395 static int
13396 mptsas_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
13397 {
13398         /*
13399          * as the driver can always deal with an error in any dma or
13400          * access handle, we can just return the fme_status value.
13401          */
13402         pci_ereport_post(dip, err, NULL);
13403         return (err->fme_status);
13404 }

13406 /*
13407  * mptsas_fm_init - initialize fma capabilities and register with IO
13408  *               fault services.
13409  */
13410 static void
13411 mptsas_fm_init(mptsas_t *mpt)
13412 {
13413         /*
13414          * Need to change iblock to priority for new MSI intr
13415          */
13416         ddi_iblock_cookie_t     fm_ibc;

13418         /* Only register with IO Fault Services if we have some capability */
13419         if (mpt->m_fm_capabilities) {
13420                 /* Adjust access and dma attributes for FMA */
13421                 mpt->m_reg_acc_attr.devacc_attr_access = DDI_FLAGERR_ACC;
13422                 mpt->m_msg_dma_attr.dma_attr_flags |= DDI_DMA_FLAGERR;
13423                 mpt->m_io_dma_attr.dma_attr_flags |= DDI_DMA_FLAGERR;

13425                 /*
13426                  * Register capabilities with IO Fault Services.
13427                  * mpt->m_fm_capabilities will be updated to indicate
13428                  * capabilities actually supported (not requested.)
13429                  */
13430                 ddi_fm_init(mpt->m_dip, &mpt->m_fm_capabilities, &fm_ibc);

13432                 /*
13433                  * Initialize pci ereport capabilities if ereport
```



```
13434                  * capable (should always be.)
13435                  */
13436                 if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities) ||
13437                     DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13438                         pci_ereport_setup(mpt->m_dip);
13439                 }

13441                 /*
13442                  * Register error callback if error callback capable.
13443                  */
13444                 if (DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13445                         ddi_fm_handler_register(mpt->m_dip,
13446                             mptsas_fm_error_cb, (void *) mpt);
13447                 }
13448         }
13449 }

13451 /*
13452  * mptsas_fm_fini - Releases fma capabilities and un-registers with IO
13453  *               fault services.
13454  *
13455  */
13456 static void
13457 mptsas_fm_fini(mptsas_t *mpt)
13458 {
13459         /* Only unregister FMA capabilities if registered */
13460         if (mpt->m_fm_capabilities) {

13462                 /*
13463                  * Un-register error callback if error callback capable.
13464                  */

13466                 if (DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13467                         ddi_fm_handler_unregister(mpt->m_dip);
13468                 }

13470                 /*
13471                  * Release any resources allocated by pci_ereport_setup()
13472                  */

13474                 if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities) ||
13475                     DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13476                         pci_ereport_teardown(mpt->m_dip);
13477                 }

13479                 /* Unregister from IO Fault Services */
13480                 ddi_fm_fini(mpt->m_dip);

13482                 /* Adjust access and dma attributes for FMA */
13483                 mpt->m_reg_acc_attr.devacc_attr_access = DDI_DEFAULT_ACC;
13484                 mpt->m_msg_dma_attr.dma_attr_flags &= ~DDI_DMA_FLAGERR;
13485                 mpt->m_io_dma_attr.dma_attr_flags &= ~DDI_DMA_FLAGERR;

13487         }
13488 }

13490 int
13491 mptsas_check_acc_handle(ddi_acc_handle_t handle)
13492 {
13493         ddi_fm_error_t  de;

13495         if (handle == NULL)
13496                 return (DDI_FAILURE);
13497         ddi_fm_acc_err_get(handle, &de, DDI_FME_VER0);
13498         return (de.fme_status);
13499 }
```

```
13501 int
13502 mptsas_check_dma_handle(ddi_dma_handle_t handle)
13503 {
13504         ddi_fm_error_t  de;

13506         if (handle == NULL)
13507                 return (DDI_FAILURE);
13508         ddi_fm_dma_err_get(handle, &de, DDI_FME_VER0);
13509         return (de.fme_status);
13510 }

13512 void
13513 mptsas_fm_ereport(mptsas_t *mpt, char *detail)
13514 {
13515         uint64_t        ena;
13516         char            buf[FM_MAX_CLASS];

13518         (void) snprintf(buf, FM_MAX_CLASS, "%s.%s", DDI_FM_DEVICE, detail);
13519         ena = fm_ena_generate(0, FM_ENA_FMT1);
13520         if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities)) {
13521                 ddi_fm_ereport_post(mpt->m_dip, buf, ena, DDI_NOSLEEP,
13522                     FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERS0, NULL);
13523         }
13524 }

13526 static int
13527 mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
13528     uint16_t *dev_handle, mptsas_target_t **pptgt)
13529 {
13530         int             rval;
13531         uint32_t        dev_info;
13532         uint64_t        sas_wwn;
13533         mptsas_phymask_t phymask;
13534         uint8_t         physport, phynum, config, disk;
13535         uint64_t        devicename;
13536         uint16_t        pdev_hdl;
13537         mptsas_target_t *tmp_tgt = NULL;
13538         uint16_t        bay_num, enclosure, io_flags;

13540         ASSERT(*pptgt == NULL);

13542         rval = mptsas_get_sas_device_page0(mpt, page_address, dev_handle,
13543             &sas_wwn, &dev_info, &physport, &phynum, &pdev_hdl,
13544             &bay_num, &enclosure, &io_flags);
13545         if (rval != DDI_SUCCESS) {
13546                 rval = DEV_INFO_FAIL_PAGE0;
13547                 return (rval);
13548         }

13550         if ((dev_info & (MPI2_SAS_DEVICE_INFO_SSP_TARGET |
13551             MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
13552             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) == NULL) {
13553                 rval = DEV_INFO_WRONG_DEVICE_TYPE;
13554                 return (rval);
13555         }

13557         /*
13558          * Check if the dev handle is for a Phys Disk. If so, set return value
13559          * and exit.  Don't add Phys Disks to hash.
13560          */
13561         for (config = 0; config < mpt->m_num_raid_configs; config++) {
13562                 for (disk = 0; disk < MPTSAS_MAX_DISKS_IN_CONFIG; disk++) {
13563                         if (*dev_handle == mpt->m_raidconfig[config].
13564                             m_physdisk_devhdl[disk]) {
13565                                 rval = DEV_INFO_PHYS_DISK;
```



```
13566                                 return (rval);
13567                         }
13568                 }
13569         }

13571         /*
13572          * Get SATA Device Name from SAS device page0 for
13573          * sata device, if device name doesn't exist, set mta_wwn to
13574          * 0 for direct attached SATA. For the device behind the expander
13575          * we still can use STP address assigned by expander.
13576          */
13577         if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
13578             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
13579                 mutex_exit(&mpt->m_mutex);
13580                 /* alloc a tmp_tgt to send the cmd */
13581                 tmp_tgt = kmem_zalloc(sizeof (struct mptsas_target),
13582                     KM_SLEEP);
13583                 tmp_tgt->m_devhdl = *dev_handle;
13584                 tmp_tgt->m_deviceinfo = dev_info;
13585                 tmp_tgt->m_qfull_retries = QFULL_RETRIES;
13586                 tmp_tgt->m_qfull_retry_interval =
13587                     drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
13588                 tmp_tgt->m_t_throttle = MAX_THROTTLE;
13589                 devicename = mptsas_get_sata_guid(mpt, tmp_tgt, 0);
13590                 kmem_free(tmp_tgt, sizeof (struct mptsas_target));
13591                 mutex_enter(&mpt->m_mutex);
13592                 if (devicename != 0 && (((devicename >> 56) & 0xf0) == 0x50)) {
13593                         sas_wwn = devicename;
13594                 } else if (dev_info & MPI2_SAS_DEVICE_INFO_DIRECT_ATTACH) {
13595                         sas_wwn = 0;
13596                 }
13597         }

13599         phymask = mptsas_physport_to_phymask(mpt, physport);
13600         *pptgt = mptsas_tgt_alloc(mpt, *dev_handle, sas_wwn,
13601             dev_info, phymask, phynum);
13602         if (*pptgt == NULL) {
13603                 mptsas_log(mpt, CE_WARN, "Failed to allocated target"
13604                     "structure!");
13605                 rval = DEV_INFO_FAIL_ALLOC;
13606                 return (rval);
13607         }
13608         (*pptgt)->m_io_flags = io_flags;
13609         (*pptgt)->m_enclosure = enclosure;
13610         (*pptgt)->m_slot_num = bay_num;
13611         return (DEV_INFO_SUCCESS);
13612 }

13614 uint64_t
13615 mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun)
13616 {
13617         uint64_t        sata_guid = 0, *pwwn = NULL;
13618         int             target = ptgt->m_devhdl;
13619         uchar_t         *inq83 = NULL;
13620         int             inq83_len = 0xFF;
13621         uchar_t         *dblk = NULL;
13622         int             inq83_retry = 3;
13623         int             rval = DDI_FAILURE;

13625         inq83   = kmem_zalloc(inq83_len, KM_SLEEP);

13627 inq83_retry:
13628         rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
13629             inq83_len, NULL, 1);
13630         if (rval != DDI_SUCCESS) {
13631                 mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
```

```
13632                     "0x83 for target:%x, lun:%x failed!", target, lun);
13633                 goto out;
13634         }
13635         /* According to SAT2, the first descriptor is logic unit name */
13636         dblk = &inq83[4];
13637         if ((dblk[1] & 0x30) != 0) {
13638                 mptsas_log(mpt, CE_WARN, "!Descriptor is not lun associated.");
13639                 goto out;
13640         }
13641         pwwn = (uint64_t *)(void *)(&dblk[4]);
13642         if ((dblk[4] & 0xf0) == 0x50) {
13643                 sata_guid = BE_64(*pwwn);
13644                 goto out;
13645         } else if (dblk[4] == 'A') {
13646                 NDBG20(("SATA drive has no NAA format GUID."));
13647                 goto out;
13648         } else {
13649                 /* The data is not ready, wait and retry */
13650                 inq83_retry--;
13651                 if (inq83_retry <= 0) {
13652                         goto out;
13653                 }
13654                 NDBG20(("The GUID is not ready, retry..."));
13655                 delay(1 * drv_usectohz(1000000));
13656                 goto inq83_retry;
13657         }
13658 out:
13659         kmem_free(inq83, inq83_len);
13660         return (sata_guid);
13661 }

13663 static int
13664 mptsas_inquiry(mptsas_t *mpt, mptsas_target_t *ptgt, int lun, uchar_t page,
13665     unsigned char *buf, int len, int *reallen, uchar_t evpd)
13666 {
13667         uchar_t                 cdb[CDB_GROUP0];
13668         struct scsi_address     ap;
13669         struct buf              *data_bp = NULL;
13670         int                     resid = 0;
13671         int                     ret = DDI_FAILURE;

13673         ASSERT(len <= 0xffff);

13675         ap.a_target = MPTSAS_INVALID_DEVHDL;
13676         ap.a_lun = (uchar_t)(lun);
13677         ap.a_hba_tran = mpt->m_tran;

13679         data_bp = scsi_alloc_consistent_buf(&ap,
13680             (struct buf *)NULL, len, B_READ, NULL_FUNC, NULL);
13681         if (data_bp == NULL) {
13682                 return (ret);
13683         }
13684         bzero(cdb, CDB_GROUP0);
13685         cdb[0] = SCMD_INQUIRY;
13686         cdb[1] = evpd;
13687         cdb[2] = page;
13688         cdb[3] = (len & 0xff00) >> 8;
13689         cdb[4] = (len & 0x00ff);
13690         cdb[5] = 0;

13692         ret = mptsas_send_scsi_cmd(mpt, &ap, ptgt, &cdb[0], CDB_GROUP0, data_bp,
13693             &resid);
13694         if (ret == DDI_SUCCESS) {
13695                 if (reallen) {
13696                         *reallen = len - resid;
13697                 }
```



```
13698                 bcopy((caddr_t)data_bp->b_un.b_addr, buf, len);
13699         }
13700         if (data_bp) {
13701                 scsi_free_consistent_buf(data_bp);
13702         }
13703         return (ret);
13704 }

13706 static int
13707 mptsas_send_scsi_cmd(mptsas_t *mpt, struct scsi_address *ap,
13708     mptsas_target_t *ptgt, uchar_t *cdb, int cdblen, struct buf *data_bp,
13709     int *resid)
13710 {
13711         struct scsi_pkt         *pktp = NULL;
13712         scsi_hba_tran_t         *tran_clone = NULL;
13713         mptsas_tgt_private_t    *tgt_private = NULL;
13714         int                     ret = DDI_FAILURE;

13716         /*
13717          * scsi_hba_tran_t->tran_tgt_private is used to pass the address
13718          * information to scsi_init_pkt, allocate a scsi_hba_tran structure
13719          * to simulate the cmds from sd
13720          */
13721         tran_clone = kmem_alloc(
13722             sizeof (scsi_hba_tran_t), KM_SLEEP);
13723         if (tran_clone == NULL) {
13724                 goto out;
13725         }
13726         bcopy((caddr_t)mpt->m_tran,
13727             (caddr_t)tran_clone, sizeof (scsi_hba_tran_t));
13728         tgt_private = kmem_alloc(
13729             sizeof (mptsas_tgt_private_t), KM_SLEEP);
13730         if (tgt_private == NULL) {
13731                 goto out;
13732         }
13733         tgt_private->t_lun = ap->a_lun;
13734         tgt_private->t_private = ptgt;
13735         tran_clone->tran_tgt_private = tgt_private;
13736         ap->a_hba_tran = tran_clone;

13738         pktp = scsi_init_pkt(ap, (struct scsi_pkt *)NULL,
13739             data_bp, cdblen, sizeof (struct scsi_arq_status),
13740             0, PKT_CONSISTENT, NULL, NULL);
13741         if (pktp == NULL) {
13742                 goto out;
13743         }
13744         bcopy(cdb, pktp->pkt_cdbp, cdblen);
13745         pktp->pkt_flags = FLAG_NOPARITY;
13746         if (scsi_poll(pktp) < 0) {
13747                 goto out;
13748         }
13749         if (((struct scsi_status *)pktp->pkt_scbp)->sts_chk) {
13750                 goto out;
13751         }
13752         if (resid != NULL) {
13753                 *resid = pktp->pkt_resid;
13754         }

13756         ret = DDI_SUCCESS;
13757 out:
13758         if (pktp) {
13759                 scsi_destroy_pkt(pktp);
13760         }
13761         if (tran_clone) {
13762                 kmem_free(tran_clone, sizeof (scsi_hba_tran_t));
13763         }
```

```
13764         if (tgt_private) {
13765                 kmem_free(tgt_private, sizeof (mptsas_tgt_private_t));
13766         }
13767         return (ret);
13768 }
13769 static int
13770 mptsas_parse_address(char *name, uint64_t *wwid, uint8_t *phy, int *lun)
13771 {
13772         char    *cp = NULL;
13773         char    *ptr = NULL;
13774         size_t  s = 0;
13775         char    *wwid_str = NULL;
13776         char    *lun_str = NULL;
13777         long    lunnum;
13778         long    phyid = -1;
13779         int     rc = DDI_FAILURE;

13781         ptr = name;
13782         ASSERT(ptr[0] == 'w' || ptr[0] == 'p');
13783         ptr++;
13784         if ((cp = strchr(ptr, ',')) == NULL) {
13785                 return (DDI_FAILURE);
13786         }

13788         wwid_str = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13789         s = (uintptr_t)cp - (uintptr_t)ptr;

13791         bcopy(ptr, wwid_str, s);
13792         wwid_str[s] = '\0';

13794         ptr = ++cp;

13796         if ((cp = strchr(ptr, '\0')) == NULL) {
13797                 goto out;
13798         }
13799         lun_str =  kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13800         s = (uintptr_t)cp - (uintptr_t)ptr;

13802         bcopy(ptr, lun_str, s);
13803         lun_str[s] = '\0';

13805         if (name[0] == 'p') {
13806                 rc = ddi_strtol(wwid_str, NULL, 0x10, &phyid);
13807         } else {
13808                 rc = scsi_wwnstr_to_wwn(wwid_str, wwid);
13809         }
13810         if (rc != DDI_SUCCESS)
13811                 goto out;

13813         if (phyid != -1) {
13814                 ASSERT(phyid < MPTSAS_MAX_PHYS);
13815                 *phy = (uint8_t)phyid;
13816         }
13817         rc = ddi_strtol(lun_str, NULL, 0x10, &lunnum);
13818         if (rc != 0)
13819                 goto out;

13821         *lun = (int)lunnum;
13822         rc = DDI_SUCCESS;
13823 out:
13824         if (wwid_str)
13825                 kmem_free(wwid_str, SCSI_MAXNAMELEN);
13826         if (lun_str)
13827                 kmem_free(lun_str, SCSI_MAXNAMELEN);

13829         return (rc);
```



```
13830 }

13832 /*
13833  * mptsas_parse_smp_name() is to parse sas wwn string
13834  * which format is "wWWN"
13835  */
13836 static int
13837 mptsas_parse_smp_name(char *name, uint64_t *wwn)
13838 {
13839         char    *ptr = name;

13841         if (*ptr != 'w') {
13842                 return (DDI_FAILURE);
13843         }

13845         ptr++;
13846         if (scsi_wwnstr_to_wwn(ptr, wwn)) {
13847                 return (DDI_FAILURE);
13848         }
13849         return (DDI_SUCCESS);
13850 }

13852 static int
13853 mptsas_bus_config(dev_info_t *pdip, uint_t flag,
13854     ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
13855 {
13856         int             ret = NDI_FAILURE;
13857         int             circ = 0;
13858         int             circ1 = 0;
13859         mptsas_t        *mpt;
13860         char            *ptr = NULL;
13861         char            *devnm = NULL;
13862         uint64_t        wwid = 0;
13863         uint8_t         phy = 0xFF;
13864         int             lun = 0;
13865         uint_t          mflags = flag;
13866         int             bconfig = TRUE;

13868         if (scsi_hba_iport_unit_address(pdip) == 0) {
13869                 return (DDI_FAILURE);
13870         }

13872         mpt = DIP2MPT(pdip);
13873         if (!mpt) {
13874                 return (DDI_FAILURE);
13875         }
13876         /*
13877          * Hold the nexus across the bus_config
13878          */
13879         ndi_devi_enter(scsi_vhci_dip, &circ);
13880         ndi_devi_enter(pdip, &circ1);
13881         switch (op) {
13882         case BUS_CONFIG_ONE:
13883                 /* parse wwid/target name out of name given */
13884                 if ((ptr = strchr((char *)arg, '@')) == NULL) {
13885                         ret = NDI_FAILURE;
13886                         break;
13887                 }
13888                 ptr++;
13889                 if (strncmp((char *)arg, "smp", 3) == 0) {
13890                         /*
13891                          * This is a SMP target device
13892                          */
13893                         ret = mptsas_parse_smp_name(ptr, &wwid);
13894                         if (ret != DDI_SUCCESS) {
13895                                 ret = NDI_FAILURE;
```

```
13896                                   break;
13897                           }
13898                           ret = mptsas_config_smp(pdip, wwid, childp);
13899                   } else if ((ptr[0] == 'w') || (ptr[0] == 'p')) {
13900                           /*
13901                            * OBP could pass down a non-canonical form
13902                            * bootpath without LUN part when LUN is 0.
13903                            * So driver need adjust the string.
13904                            */
13905                           if (strchr(ptr, ',') == NULL) {
13906                                   devnm = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13907                                   (void) sprintf(devnm, "%s,0", (char *)arg);
13908                                   ptr = strchr(devnm, '@');
13909                                   ptr++;
13910                           }

13912                           /*
13913                            * The device path is wWWID format and the device
13914                            * is not SMP target device.
13915                            */
13916                           ret = mptsas_parse_address(ptr, &wwid, &phy, &lun);
13917                           if (ret != DDI_SUCCESS) {
13918                                   ret = NDI_FAILURE;
13919                                   break;
13920                           }
13921                           *childp = NULL;
13922                           if (ptr[0] == 'w') {
13923                                   ret = mptsas_config_one_addr(pdip, wwid,
13924                                       lun, childp);
13925                           } else if (ptr[0] == 'p') {
13926                                   ret = mptsas_config_one_phy(pdip, phy, lun,
13927                                       childp);
13928                           }

13930                           /*
13931                            * If this is CD/DVD device in OBP path, the
13932                            * ndi_busop_bus_config can be skipped as config one
13933                            * operation is done above.
13934                            */
13935                           if ((ret == NDI_SUCCESS) && (*childp != NULL) &&
13936                               (strcmp(ddi_node_name(*childp), "cdrom") == 0) &&
13937                               (strncmp((char *)arg, "disk", 4) == 0)) {
13938                                   bconfig = FALSE;
13939                                   ndi_hold_devi(*childp);
13940                           }
13941                   } else {
13942                           ret = NDI_FAILURE;
13943                           break;
13944                   }

13946                   /*
13947                    * DDI group instructed us to use this flag.
13948                    */
13949                   mflags |= NDI_MDI_FALLBACK;
13950                   break;
13951           case BUS_CONFIG_DRIVER:
13952           case BUS_CONFIG_ALL:
13953                   mptsas_config_all(pdip);
13954                   ret = NDI_SUCCESS;
13955                   break;
13956           }

13958           if ((ret == NDI_SUCCESS) && bconfig) {
13959                   ret = ndi_busop_bus_config(pdip, mflags, op,
13960                       (devnm == NULL) ? arg : devnm, childp, 0);
13961           }
```



```
13963           ndi_devi_exit(pdip, circ1);
13964           ndi_devi_exit(scsi_vhci_dip, circ);
13965           if (devnm != NULL)
13966                   kmem_free(devnm, SCSI_MAXNAMELEN);
13967           return (ret);
13968   }

13970   static int
13971   mptsas_probe_lun(dev_info_t *pdip, int lun, dev_info_t **dip,
13972       mptsas_target_t *ptgt)
13973   {
13974           int                     rval = DDI_FAILURE;
13975           struct scsi_inquiry     *sd_inq = NULL;
13976           mptsas_t                *mpt = DIP2MPT(pdip);

13978           sd_inq = (struct scsi_inquiry *)kmem_alloc(SUN_INQSIZE, KM_SLEEP);

13980           rval = mptsas_inquiry(mpt, ptgt, lun, 0, (uchar_t *)sd_inq,
13981               SUN_INQSIZE, 0, (uchar_t)0);

13983           if ((rval == DDI_SUCCESS) && MPTSAS_VALID_LUN(sd_inq)) {
13984                   rval = mptsas_create_lun(pdip, sd_inq, dip, ptgt, lun);
13985           } else {
13986                   rval = DDI_FAILURE;
13987           }

13989           kmem_free(sd_inq, SUN_INQSIZE);
13990           return (rval);
13991   }

13993   static int
13994   mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
13995       dev_info_t **lundip)
13996   {
13997           int             rval;
13998           mptsas_t                *mpt = DIP2MPT(pdip);
13999           int             phymask;
14000           mptsas_target_t *ptgt = NULL;

14002           /*
14003            * Get the physical port associated to the iport
14004            */
14005           phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14006               "phymask", 0);

14008           ptgt = mptsas_wwid_to_ptgt(mpt, phymask, sasaddr);
14009           if (ptgt == NULL) {
14010                   /*
14011                    * didn't match any device by searching
14012                    */
14013                   return (DDI_FAILURE);
14014           }
14015           /*
14016            * If the LUN already exists and the status is online,
14017            * we just return the pointer to dev_info_t directly.
14018            * For the mdi_pathinfo node, we'll handle it in
14019            * mptsas_create_virt_lun()
14020            * TODO should be also in mptsas_handle_dr
14021            */

14023           *lundip = mptsas_find_child_addr(pdip, sasaddr, lun);
14024           if (*lundip != NULL) {
14025                   /*
14026                    * TODO Another senario is, we hotplug the same disk
14027                    * on the same slot, the devhdl changed, is this
```

```
14028                  * possible?
14029                  * tgt_private->t_private != ptgt
14030                  */
14031                 if (sasaddr != ptgt->m_addr.mta_wwn) {
14032                         /*
14033                          * The device has changed although the devhdl is the
14034                          * same (Enclosure mapping mode, change drive on the
14035                          * same slot)
14036                          */
14037                         return (DDI_FAILURE);
14038                 }
14039                 return (DDI_SUCCESS);
14040         }

14042         if (phymask == 0) {
14043                 /*
14044                  * Configure IR volume
14045                  */
14046                 rval =  mptsas_config_raid(pdip, ptgt->m_devhdl, lundip);
14047                 return (rval);
14048         }
14049         rval = mptsas_probe_lun(pdip, lun, lundip, ptgt);

14051         return (rval);
14052 }

14054 static int
14055 mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
14056     dev_info_t **lundip)
14057 {
14058         int             rval;
14059         mptsas_t        *mpt = DIP2MPT(pdip);
14060         mptsas_phymask_t phymask;
14061         mptsas_target_t *ptgt = NULL;

14063         /*
14064          * Get the physical port associated to the iport
14065          */
14066         phymask = (mptsas_phymask_t)ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14067             "phymask", 0);

14069         ptgt = mptsas_phy_to_tgt(mpt, phymask, phy);
14070         if (ptgt == NULL) {
14071                 /*
14072                  * didn't match any device by searching
14073                  */
14074                 return (DDI_FAILURE);
14075         }

14077         /*
14078          * If the LUN already exists and the status is online,
14079          * we just return the pointer to dev_info_t directly.
14080          * For the mdi_pathinfo node, we'll handle it in
14081          * mptsas_create_virt_lun().
14082          */

14084         *lundip = mptsas_find_child_phy(pdip, phy);
14085         if (*lundip != NULL) {
14086                 return (DDI_SUCCESS);
14087         }

14089         rval = mptsas_probe_lun(pdip, lun, lundip, ptgt);

14091         return (rval);
14092 }
```



```
14094 static int
14095 mptsas_retrieve_lundata(int lun_cnt, uint8_t *buf, uint16_t *lun_num,
14096     uint8_t *lun_addr_type)
14097 {
14098         uint32_t        lun_idx = 0;

14100         ASSERT(lun_num != NULL);
14101         ASSERT(lun_addr_type != NULL);

14103         lun_idx = (lun_cnt + 1) * MPTSAS_SCSI_REPORTLUNS_ADDRESS_SIZE;
14104         /* determine report luns addressing type */
14105         switch (buf[lun_idx] & MPTSAS_SCSI_REPORTLUNS_ADDRESS_MASK) {
14106                 /*
14107                  * Vendors in the field have been found to be concatenating
14108                  * bus/target/lun to equal the complete lun value instead
14109                  * of switching to flat space addressing
14110                  */
14111                 /* 00b - peripheral device addressing method */
14112         case MPTSAS_SCSI_REPORTLUNS_ADDRESS_PERIPHERAL:
14113                 /* FALLTHRU */
14114                 /* 10b - logical unit addressing method */
14115         case MPTSAS_SCSI_REPORTLUNS_ADDRESS_LOGICAL_UNIT:
14116                 /* FALLTHRU */
14117                 /* 01b - flat space addressing method */
14118         case MPTSAS_SCSI_REPORTLUNS_ADDRESS_FLAT_SPACE:
14119                 /* byte0 bit0-5=msb lun byte1 bit0-7=lsb lun */
14120                 *lun_addr_type = (buf[lun_idx] &
14121                     MPTSAS_SCSI_REPORTLUNS_ADDRESS_MASK) >> 6;
14122                 *lun_num = (buf[lun_idx] & 0x3F) << 8;
14123                 *lun_num |= buf[lun_idx + 1];
14124                 return (DDI_SUCCESS);
14125         default:
14126                 return (DDI_FAILURE);
14127         }
14128 }

14130 static int
14131 mptsas_config_luns(dev_info_t *pdip, mptsas_target_t *ptgt)
14132 {
14133         struct buf              *repluns_bp = NULL;
14134         struct scsi_address     ap;
14135         uchar_t                 cdb[CDB_GROUP5];
14136         int                     ret = DDI_FAILURE;
14137         int                     retry = 0;
14138         int                     lun_list_len = 0;
14139         uint16_t                lun_num = 0;
14140         uint8_t                 lun_addr_type = 0;
14141         uint32_t                lun_cnt = 0;
14142         uint32_t                lun_total = 0;
14143         dev_info_t              *cdip = NULL;
14144         uint16_t                *saved_repluns = NULL;
14145         char                    *buffer = NULL;
14146         int                     buf_len = 128;
14147         mptsas_t                *mpt = DIP2MPT(pdip);
14148         uint64_t                sas_wwn = 0;
14149         uint8_t                 phy = 0xFF;
14150         uint32_t                dev_info = 0;

14152         mutex_enter(&mpt->m_mutex);
14153         sas_wwn = ptgt->m_addr.mta_wwn;
14154         phy = ptgt->m_phynum;
14155         dev_info = ptgt->m_deviceinfo;
14156         mutex_exit(&mpt->m_mutex);

14158         if (sas_wwn == 0) {
14159                 /*
```

```
14160                  * It's a SATA without Device Name
14161                  * So don't try multi-LUNs
14162                  */
14163                 if (mptsas_find_child_phy(pdip, phy)) {
14164                         return (DDI_SUCCESS);
14165                 } else {
14166                         /*
14167                          * need configure and create node
14168                          */
14169                         return (DDI_FAILURE);
14170                 }
14171         }

14173         /*
14174          * WWN (SAS address or Device Name exist)
14175          */
14176         if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14177             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14178                 /*
14179                  * SATA device with Device Name
14180                  * So don't try multi-LUNs
14181                  */
14182                 if (mptsas_find_child_addr(pdip, sas_wwn, 0)) {
14183                         return (DDI_SUCCESS);
14184                 } else {
14185                         return (DDI_FAILURE);
14186                 }
14187         }

14189         do {
14190                 ap.a_target = MPTSAS_INVALID_DEVHDL;
14191                 ap.a_lun = 0;
14192                 ap.a_hba_tran = mpt->m_tran;
14193                 repluns_bp = scsi_alloc_consistent_buf(&ap,
14194                     (struct buf *)NULL, buf_len, B_READ, NULL_FUNC, NULL);
14195                 if (repluns_bp == NULL) {
14196                         retry++;
14197                         continue;
14198                 }
14199                 bzero(cdb, CDB_GROUP5);
14200                 cdb[0] = SCMD_REPORT_LUNS;
14201                 cdb[6] = (buf_len & 0xff000000) >> 24;
14202                 cdb[7] = (buf_len & 0x00ff0000) >> 16;
14203                 cdb[8] = (buf_len & 0x0000ff00) >> 8;
14204                 cdb[9] = (buf_len & 0x000000ff);

14206                 ret = mptsas_send_scsi_cmd(mpt, &ap, ptgt, &cdb[0], CDB_GROUP5,
14207                     repluns_bp, NULL);
14208                 if (ret != DDI_SUCCESS) {
14209                         scsi_free_consistent_buf(repluns_bp);
14210                         retry++;
14211                         continue;
14212                 }
14213                 lun_list_len = BE_32(*(int *)((void *)(
14214                     repluns_bp->b_un.b_addr)));
14215                 if (buf_len >= lun_list_len + 8) {
14216                         ret = DDI_SUCCESS;
14217                         break;
14218                 }
14219                 scsi_free_consistent_buf(repluns_bp);
14220                 buf_len = lun_list_len + 8;

14222         } while (retry < 3);

14224         if (ret != DDI_SUCCESS)
14225                 return (ret);
```



```
14226         buffer = (char *)repluns_bp->b_un.b_addr;
14227         /*
14228          * find out the number of luns returned by the SCSI ReportLun call
14229          * and allocate buffer space
14230          */
14231         lun_total = lun_list_len / MPTSAS_SCSI_REPORTLUNS_ADDRESS_SIZE;
14232         saved_repluns = kmem_zalloc(sizeof (uint16_t) * lun_total, KM_SLEEP);
14233         if (saved_repluns == NULL) {
14234                 scsi_free_consistent_buf(repluns_bp);
14235                 return (DDI_FAILURE);
14236         }
14237         for (lun_cnt = 0; lun_cnt < lun_total; lun_cnt++) {
14238                 if (mptsas_retrieve_lundata(lun_cnt, (uint8_t *)(buffer),
14239                     &lun_num, &lun_addr_type) != DDI_SUCCESS) {
14240                         continue;
14241                 }
14242                 saved_repluns[lun_cnt] = lun_num;
14243                 if (cdip = mptsas_find_child_addr(pdip, sas_wwn, lun_num))
14244                         ret = DDI_SUCCESS;
14245                 else
14246                         ret = mptsas_probe_lun(pdip, lun_num, &cdip,
14247                             ptgt);
14248                 if ((ret == DDI_SUCCESS) && (cdip != NULL)) {
14249                         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip,
14250                             MPTSAS_DEV_GONE);
14251                 }
14252         }
14253         mptsas_offline_missed_luns(pdip, saved_repluns, lun_total, ptgt);
14254         kmem_free(saved_repluns, sizeof (uint16_t) * lun_total);
14255         scsi_free_consistent_buf(repluns_bp);
14256         return (DDI_SUCCESS);
14257 }

14259 static int
14260 mptsas_config_raid(dev_info_t *pdip, uint16_t target, dev_info_t **dip)
14261 {
14262         int                     rval = DDI_FAILURE;
14263         struct scsi_inquiry     *sd_inq = NULL;
14264         mptsas_t                *mpt = DIP2MPT(pdip);
14265         mptsas_target_t         *ptgt = NULL;

14267         mutex_enter(&mpt->m_mutex);
14268         ptgt = refhash_linear_search(mpt->m_targets,
14269             mptsas_target_eval_devhdl, &target);
14270         mutex_exit(&mpt->m_mutex);
14271         if (ptgt == NULL) {
14272                 mptsas_log(mpt, CE_WARN, "Volume with VolDevHandle of 0x%x "
14273                     "not found.", target);
14274                 return (rval);
14275         }

14277         sd_inq = (struct scsi_inquiry *)kmem_alloc(SUN_INQSIZE, KM_SLEEP);
14278         rval = mptsas_inquiry(mpt, ptgt, 0, 0, (uchar_t *)sd_inq,
14279             SUN_INQSIZE, 0, (uchar_t)0);

14281         if ((rval == DDI_SUCCESS) && MPTSAS_VALID_LUN(sd_inq)) {
14282                 rval = mptsas_create_phys_lun(pdip, sd_inq, NULL, dip, ptgt,
14283                     0);
14284         } else {
14285                 rval = DDI_FAILURE;
14286         }

14288         kmem_free(sd_inq, SUN_INQSIZE);
14289         return (rval);
14290 }
```

```
14292 /*
14293  * configure all RAID volumes for virtual iport
14294  */
14295 static void
14296 mptsas_config_all_viport(dev_info_t *pdip)
14297 {
14298         mptsas_t        *mpt = DIP2MPT(pdip);
14299         int             config, vol;
14300         int             target;
14301         dev_info_t      *lundip = NULL;

14303         /*
14304          * Get latest RAID info and search for any Volume DevHandles.  If any
14305          * are found, configure the volume.
14306          */
14307         mutex_enter(&mpt->m_mutex);
14308         for (config = 0; config < mpt->m_num_raid_configs; config++) {
14309                 for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
14310                         if (mpt->m_raidconfig[config].m_raidvol[vol].m_israid
14311                             == 1) {
14312                                 target = mpt->m_raidconfig[config].
14313                                     m_raidvol[vol].m_raidhandle;
14314                                 mutex_exit(&mpt->m_mutex);
14315                                 (void) mptsas_config_raid(pdip, target,
14316                                     &lundip);
14317                                 mutex_enter(&mpt->m_mutex);
14318                         }
14319                 }
14320         }
14321         mutex_exit(&mpt->m_mutex);
14322 }

14324 static void
14325 mptsas_offline_missed_luns(dev_info_t *pdip, uint16_t *repluns,
14326     int lun_cnt, mptsas_target_t *ptgt)
14327 {
14328         dev_info_t      *child = NULL, *savechild = NULL;
14329         mdi_pathinfo_t  *pip = NULL, *savepip = NULL;
14330         uint64_t        sas_wwn, wwid;
14331         uint8_t         phy;
14332         int             lun;
14333         int             i;
14334         int             find;
14335         char            *addr;
14336         char            *nodename;
14337         mptsas_t        *mpt = DIP2MPT(pdip);

14339         mutex_enter(&mpt->m_mutex);
14340         wwid = ptgt->m_addr.mta_wwn;
14341         mutex_exit(&mpt->m_mutex);

14343         child = ddi_get_child(pdip);
14344         while (child) {
14345                 find = 0;
14346                 savechild = child;
14347                 child = ddi_get_next_sibling(child);

14349                 nodename = ddi_node_name(savechild);
14350                 if (strcmp(nodename, "smp") == 0) {
14351                         continue;
14352                 }

14354                 addr = ddi_get_name_addr(savechild);
14355                 if (addr == NULL) {
14356                         continue;
14357                 }
```



```
14359                 if (mptsas_parse_address(addr, &sas_wwn, &phy, &lun) !=
14360                     DDI_SUCCESS) {
14361                         continue;
14362                 }

14364                 if (wwid == sas_wwn) {
14365                         for (i = 0; i < lun_cnt; i++) {
14366                                 if (repluns[i] == lun) {
14367                                         find = 1;
14368                                         break;
14369                                 }
14370                         }
14371                 } else {
14372                         continue;
14373                 }
14374                 if (find == 0) {
14375                         /*
14376                          * The lun has not been there already
14377                          */
14378                         (void) mptsas_offline_lun(pdip, savechild, NULL,
14379                             NDI_DEVI_REMOVE);
14380                 }
14381         }

14383         pip = mdi_get_next_client_path(pdip, NULL);
14384         while (pip) {
14385                 find = 0;
14386                 savepip = pip;
14387                 addr = MDI_PI(pip)->pi_addr;

14389                 pip = mdi_get_next_client_path(pdip, pip);

14391                 if (addr == NULL) {
14392                         continue;
14393                 }

14395                 if (mptsas_parse_address(addr, &sas_wwn, &phy,
14396                     &lun) != DDI_SUCCESS) {
14397                         continue;
14398                 }

14400                 if (sas_wwn == wwid) {
14401                         for (i = 0; i < lun_cnt; i++) {
14402                                 if (repluns[i] == lun) {
14403                                         find = 1;
14404                                         break;
14405                                 }
14406                         }
14407                 } else {
14408                         continue;
14409                 }

14411                 if (find == 0) {
14412                         /*
14413                          * The lun has not been there already
14414                          */
14415                         (void) mptsas_offline_lun(pdip, NULL, savepip,
14416                             NDI_DEVI_REMOVE);
14417                 }
14418         }
14419 }

14421 void
14422 mptsas_update_hashtab(struct mptsas *mpt)
14423 {
```

```
14424         uint32_t        page_address;
14425         int             rval = 0;
14426         uint16_t        dev_handle;
14427         mptsas_target_t *ptgt = NULL;
14428         mptsas_smp_t    smp_node;

14430         /*
14431          * Get latest RAID info.
14432          */
14433         (void) mptsas_get_raid_info(mpt);

14435         dev_handle = mpt->m_smp_devhdl;
14436         for (; mpt->m_done_traverse_smp == 0; ) {
14437                 page_address = (MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL &
14438                     MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)dev_handle;
14439                 if (mptsas_get_sas_expander_page0(mpt, page_address, &smp_node)
14440                     != DDI_SUCCESS) {
14441                         break;
14442                 }
14443                 mpt->m_smp_devhdl = dev_handle = smp_node.m_devhdl;
14444                 (void) mptsas_smp_alloc(mpt, &smp_node);
14445         }

14447         /*
14448          * Config target devices
14449          */
14450         dev_handle = mpt->m_dev_handle;

14452         /*
14453          * Do loop to get sas device page 0 by GetNextHandle till the
14454          * the last handle. If the sas device is a SATA/SSP target,
14455          * we try to config it.
14456          */
14457         for (; mpt->m_done_traverse_dev == 0; ) {
14458                 ptgt = NULL;
14459                 page_address =
14460                     (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
14461                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14462                     (uint32_t)dev_handle;
14463                 rval = mptsas_get_target_device_info(mpt, page_address,
14464                     &dev_handle, &ptgt);
14465                 if ((rval == DEV_INFO_FAIL_PAGE0) ||
14466                     (rval == DEV_INFO_FAIL_ALLOC)) {
14467                         break;
14468                 }

14470                 mpt->m_dev_handle = dev_handle;
14471         }

14473 }

14475 void
14476 mptsas_update_driver_data(struct mptsas *mpt)
14477 {
14478         mptsas_target_t *tp;
14479         mptsas_smp_t *sp;

14481         ASSERT(MUTEX_HELD(&mpt->m_mutex));

14483         /*
14484          * TODO after hard reset, update the driver data structures
14485          * 1. update port/phymask mapping table mpt->m_phy_info
14486          * 2. invalid all the entries in hash table
14487          *    m_devhdl = 0xffff and m_deviceinfo = 0
14488          * 3. call sas_device_page/expander_page to update hash table
14489          */
```



```
14490         mptsas_update_phymask(mpt);

14492         /*
14493          * Remove all the devhdls for existing entries but leave their
14494          * addresses alone.  In update_hashtab() below, we'll find all
14495          * targets that are still present and reassociate them with
14496          * their potentially new devhdls.  Leaving the targets around in
14497          * this fashion allows them to be used on the tx waitq even
14498          * while IOC reset is occurring.
14499          */
14500         for (tp = refhash_first(mpt->m_targets); tp != NULL;
14501             tp = refhash_next(mpt->m_targets, tp)) {
14502                 tp->m_devhdl = MPTSAS_INVALID_DEVHDL;
14503                 tp->m_deviceinfo = 0;
14504                 tp->m_dr_flag = MPTSAS_DR_INACTIVE;
14505         }
14506         for (sp = refhash_first(mpt->m_smp_targets); sp != NULL;
14507             sp = refhash_next(mpt->m_smp_targets, sp)) {
14508                 sp->m_devhdl = MPTSAS_INVALID_DEVHDL;
14509                 sp->m_deviceinfo = 0;
14510         }
14511         mpt->m_done_traverse_dev = 0;
14512         mpt->m_done_traverse_smp = 0;
14513         mpt->m_dev_handle = mpt->m_smp_devhdl = MPTSAS_INVALID_DEVHDL;
14514         mptsas_update_hashtab(mpt);
14515 }

14517 static void
14518 mptsas_config_all(dev_info_t *pdip)
14519 {
14520         dev_info_t      *smpdip = NULL;
14521         mptsas_t        *mpt = DIP2MPT(pdip);
14522         int             phymask = 0;
14523         mptsas_phymask_t phy_mask;
14524         mptsas_target_t *ptgt = NULL;
14525         mptsas_smp_t    *psmp;

14527         /*
14528          * Get the phymask associated to the iport
14529          */
14530         phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14531             "phymask", 0);

14533         /*
14534          * Enumerate RAID volumes here (phymask == 0).
14535          */
14536         if (phymask == 0) {
14537                 mptsas_config_all_viport(pdip);
14538                 return;
14539         }

14541         mutex_enter(&mpt->m_mutex);

14543         if (!mpt->m_done_traverse_dev || !mpt->m_done_traverse_smp) {
14544                 mptsas_update_hashtab(mpt);
14545         }

14547         for (psmp = refhash_first(mpt->m_smp_targets); psmp != NULL;
14548             psmp = refhash_next(mpt->m_smp_targets, psmp)) {
14549                 phy_mask = psmp->m_addr.mta_phymask;
14550                 if (phy_mask == phymask) {
14551                         smpdip = NULL;
14552                         mutex_exit(&mpt->m_mutex);
14553                         (void) mptsas_online_smp(pdip, psmp, &smpdip);
14554                         mutex_enter(&mpt->m_mutex);
14555                 }
```

```
14556         }

14558         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
14559             ptgt = refhash_next(mpt->m_targets, ptgt)) {
14560                 phy_mask = ptgt->m_addr.mta_phymask;
14561                 if (phy_mask == phymask) {
14562                         mutex_exit(&mpt->m_mutex);
14563                         (void) mptsas_config_target(pdip, ptgt);
14564                         mutex_enter(&mpt->m_mutex);
14565                 }
14566         }
14567         mutex_exit(&mpt->m_mutex);
14568 }

14570 static int
14571 mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt)
14572 {
14573         int             rval = DDI_FAILURE;
14574         dev_info_t      *tdip;

14576         rval = mptsas_config_luns(pdip, ptgt);
14577         if (rval != DDI_SUCCESS) {
14578                 /*
14579                  * The return value means the SCMD_REPORT_LUNS
14580                  * did not execute successfully. The target maybe
14581                  * doesn't support such command.
14582                  */
14583                 rval = mptsas_probe_lun(pdip, 0, &tdip, ptgt);
14584         }
14585         return (rval);
14586 }

14588 /*
14589  * Return fail if not all the childs/paths are freed.
14590  * if there is any path under the HBA, the return value will be always fail
14591  * because we didn't call mdi_pi_free for path
14592  */
14593 static int
14594 mptsas_offline_target(dev_info_t *pdip, char *name)
14595 {
14596         dev_info_t              *child = NULL, *prechild = NULL;
14597         mdi_pathinfo_t          *pip = NULL, *savepip = NULL;
14598         int                     tmp_rval, rval = DDI_SUCCESS;
14599         char                    *addr, *cp;
14600         size_t                  s;
14601         mptsas_t                *mpt = DIP2MPT(pdip);

14603         child = ddi_get_child(pdip);
14604         while (child) {
14605                 addr = ddi_get_name_addr(child);
14606                 prechild = child;
14607                 child = ddi_get_next_sibling(child);

14609                 if (addr == NULL) {
14610                         continue;
14611                 }
14612                 if ((cp = strchr(addr, ',')) == NULL) {
14613                         continue;
14614                 }

14616                 s = (uintptr_t)cp - (uintptr_t)addr;

14618                 if (strncmp(addr, name, s) != 0) {
14619                         continue;
14620                 }
```



```
14622                 tmp_rval = mptsas_offline_lun(pdip, prechild, NULL,
14623                     NDI_DEVI_REMOVE);
14624                 if (tmp_rval != DDI_SUCCESS) {
14625                         rval = DDI_FAILURE;
14626                         if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
14627                             prechild, MPTSAS_DEV_GONE) !=
14628                             DDI_PROP_SUCCESS) {
14629                                 mptsas_log(mpt, CE_WARN, "mptsas driver "
14630                                     "unable to create property for "
14631                                     "SAS %s (MPTSAS_DEV_GONE)", addr);
14632                         }
14633                 }
14634         }

14636         pip = mdi_get_next_client_path(pdip, NULL);
14637         while (pip) {
14638                 addr = MDI_PI(pip)->pi_addr;
14639                 savepip = pip;
14640                 pip = mdi_get_next_client_path(pdip, pip);
14641                 if (addr == NULL) {
14642                         continue;
14643                 }

14645                 if ((cp = strchr(addr, ',')) == NULL) {
14646                         continue;
14647                 }

14649                 s = (uintptr_t)cp - (uintptr_t)addr;

14651                 if (strncmp(addr, name, s) != 0) {
14652                         continue;
14653                 }

14655                 (void) mptsas_offline_lun(pdip, NULL, savepip,
14656                     NDI_DEVI_REMOVE);
14657                 /*
14658                  * driver will not invoke mdi_pi_free, so path will not
14659                  * be freed forever, return DDI_FAILURE.
14660                  */
14661                 rval = DDI_FAILURE;
14662         }
14663         return (rval);
14664 }

14666 static int
14667 mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,
14668     mdi_pathinfo_t *rpip, uint_t flags)
14669 {
14670         int             rval = DDI_FAILURE;
14671         char            *devname;
14672         dev_info_t      *cdip, *parent;

14674         if (rpip != NULL) {
14675                 parent = scsi_vhci_dip;
14676                 cdip = mdi_pi_get_client(rpip);
14677         } else if (rdip != NULL) {
14678                 parent = pdip;
14679                 cdip = rdip;
14680         } else {
14681                 return (DDI_FAILURE);
14682         }

14684         /*
14685          * Make sure node is attached otherwise
14686          * it won't have related cache nodes to
14687          * clean up.  i_ddi_devi_attached is
```

```
14688          * similiar to i_ddi_node_state(cdip) >=
14689          * DS_ATTACHED.
14690          */
14691         if (i_ddi_devi_attached(cdip)) {

14693                 /* Get full devname */
14694                 devname = kmem_alloc(MAXNAMELEN + 1, KM_SLEEP);
14695                 (void) ddi_deviname(cdip, devname);
14696                 /* Clean cache */
14697                 (void) devfs_clean(parent, devname + 1,
14698                     DV_CLEAN_FORCE);
14699                 kmem_free(devname, MAXNAMELEN + 1);
14700         }
14701         if (rpip != NULL) {
14702                 if (MDI_PI_IS_OFFLINE(rpip)) {
14703                         rval = DDI_SUCCESS;
14704                 } else {
14705                         rval = mdi_pi_offline(rpip, 0);
14706                 }
14707         } else {
14708                 rval = ndi_devi_offline(cdip, flags);
14709         }

14711         return (rval);
14712 }

14714 static dev_info_t *
14715 mptsas_find_smp_child(dev_info_t *parent, char *str_wwn)
14716 {
14717         dev_info_t      *child = NULL;
14718         char            *smp_wwn = NULL;

14720         child = ddi_get_child(parent);
14721         while (child) {
14722                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, child,
14723                     DDI_PROP_DONTPASS, SMP_WWN, &smp_wwn)
14724                     != DDI_SUCCESS) {
14725                         child = ddi_get_next_sibling(child);
14726                         continue;
14727                 }

14729                 if (strcmp(smp_wwn, str_wwn) == 0) {
14730                         ddi_prop_free(smp_wwn);
14731                         break;
14732                 }
14733                 child = ddi_get_next_sibling(child);
14734                 ddi_prop_free(smp_wwn);
14735         }
14736         return (child);
14737 }

14739 static int
14740 mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node, uint_t flags)
14741 {
14742         int             rval = DDI_FAILURE;
14743         char            *devname;
14744         char            wwn_str[MPTSAS_WWN_STRLEN];
14745         dev_info_t      *cdip;

14747         (void) sprintf(wwn_str, "%"PRIx64, smp_node->m_addr.mta_wwn);

14749         cdip = mptsas_find_smp_child(pdip, wwn_str);

14751         if (cdip == NULL)
14752                 return (DDI_SUCCESS);
```



```
14754         /*
14755          * Make sure node is attached otherwise
14756          * it won't have related cache nodes to
14757          * clean up.  i_ddi_devi_attached is
14758          * similiar to i_ddi_node_state(cdip) >=
14759          * DS_ATTACHED.
14760          */
14761         if (i_ddi_devi_attached(cdip)) {

14763                 /* Get full devname */
14764                 devname = kmem_alloc(MAXNAMELEN + 1, KM_SLEEP);
14765                 (void) ddi_deviname(cdip, devname);
14766                 /* Clean cache */
14767                 (void) devfs_clean(pdip, devname + 1,
14768                     DV_CLEAN_FORCE);
14769                 kmem_free(devname, MAXNAMELEN + 1);
14770         }

14772         rval = ndi_devi_offline(cdip, flags);

14774         return (rval);
14775 }

14777 static dev_info_t *
14778 mptsas_find_child(dev_info_t *pdip, char *name)
14779 {
14780         dev_info_t      *child = NULL;
14781         char            *rname = NULL;
14782         int             rval = DDI_FAILURE;

14784         rname = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

14786         child = ddi_get_child(pdip);
14787         while (child) {
14788                 rval = mptsas_name_child(child, rname, SCSI_MAXNAMELEN);
14789                 if (rval != DDI_SUCCESS) {
14790                         child = ddi_get_next_sibling(child);
14791                         bzero(rname, SCSI_MAXNAMELEN);
14792                         continue;
14793                 }

14795                 if (strcmp(rname, name) == 0) {
14796                         break;
14797                 }
14798                 child = ddi_get_next_sibling(child);
14799                 bzero(rname, SCSI_MAXNAMELEN);
14800         }

14802         kmem_free(rname, SCSI_MAXNAMELEN);

14804         return (child);
14805 }


14808 static dev_info_t *
14809 mptsas_find_child_addr(dev_info_t *pdip, uint64_t sasaddr, int lun)
14810 {
14811         dev_info_t      *child = NULL;
14812         char            *name = NULL;
14813         char            *addr = NULL;

14815         name = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14816         addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14817         (void) sprintf(name, "%016"PRIx64, sasaddr);
14818         (void) sprintf(addr, "w%s,%x", name, lun);
14819         child = mptsas_find_child(pdip, addr);
```

```
14820         kmem_free(name, SCSI_MAXNAMELEN);
14821         kmem_free(addr, SCSI_MAXNAMELEN);
14822         return (child);
14823 }

14825 static dev_info_t *
14826 mptsas_find_child_phy(dev_info_t *pdip, uint8_t phy)
14827 {
14828         dev_info_t      *child;
14829         char            *addr;

14831         addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14832         (void) sprintf(addr, "p%x,0", phy);
14833         child = mptsas_find_child(pdip, addr);
14834         kmem_free(addr, SCSI_MAXNAMELEN);
14835         return (child);
14836 }

14838 static mdi_pathinfo_t *
14839 mptsas_find_path_phy(dev_info_t *pdip, uint8_t phy)
14840 {
14841         mdi_pathinfo_t  *path;
14842         char            *addr = NULL;

14844         addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14845         (void) sprintf(addr, "p%x,0", phy);
14846         path = mdi_pi_find(pdip, NULL, addr);
14847         kmem_free(addr, SCSI_MAXNAMELEN);
14848         return (path);
14849 }

14851 static mdi_pathinfo_t *
14852 mptsas_find_path_addr(dev_info_t *parent, uint64_t sasaddr, int lun)
14853 {
14854         mdi_pathinfo_t  *path;
14855         char            *name = NULL;
14856         char            *addr = NULL;

14858         name = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14859         addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14860         (void) sprintf(name, "%016"PRIx64, sasaddr);
14861         (void) sprintf(addr, "w%s,%x", name, lun);
14862         path = mdi_pi_find(parent, NULL, addr);
14863         kmem_free(name, SCSI_MAXNAMELEN);
14864         kmem_free(addr, SCSI_MAXNAMELEN);

14866         return (path);
14867 }

14869 static int
14870 mptsas_create_lun(dev_info_t *pdip, struct scsi_inquiry *sd_inq,
14871     dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14872 {
14873         int                     i = 0;
14874         uchar_t                 *inq83 = NULL;
14875         int                     inq83_len1 = 0xFF;
14876         int                     inq83_len = 0;
14877         int                     rval = DDI_FAILURE;
14878         ddi_devid_t             devid;
14879         char                    *guid = NULL;
14880         int                     target = ptgt->m_devhdl;
14881         mdi_pathinfo_t          *pip = NULL;
14882         mptsas_t                *mpt = DIP2MPT(pdip);

14884         /*
14885          * For DVD/CD ROM and tape devices and optical
```



```
14886          * devices, we won't try to enumerate them under
14887          * scsi_vhci, so no need to try page83
14888          */
14889         if (sd_inq && (sd_inq->inq_dtype == DTYPE_RODIRECT ||
14890             sd_inq->inq_dtype == DTYPE_OPTICAL ||
14891             sd_inq->inq_dtype == DTYPE_ESI))
14892                 goto create_lun;

14894         /*
14895          * The LCA returns good SCSI status, but corrupt page 83 data the first
14896          * time it is queried. The solution is to keep trying to request page83
14897          * and verify the GUID is not (DDI_NOT_WELL_FORMED) in
14898          * mptsas_inq83_retry_timeout seconds. If the timeout expires, driver
14899          * give up to get VPD page at this stage and fail the enumeration.
14900          */

14902         inq83   = kmem_zalloc(inq83_len1, KM_SLEEP);

14904         for (i = 0; i < mptsas_inq83_retry_timeout; i++) {
14905                 rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
14906                     inq83_len1, &inq83_len, 1);
14907                 if (rval != 0) {
14908                         mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
14909                             "0x83 for target:%x, lun:%x failed!", target, lun);
14910                         if (mptsas_physical_bind_failed_page_83 != B_FALSE)
14911                                 goto create_lun;
14912                         goto out;
14913                 }
14914                 /*
14915                  * create DEVID from inquiry data
14916                  */
14917                 if ((rval = ddi_devid_scsi_encode(
14918                     DEVID_SCSI_ENCODE_VERSION_LATEST, NULL, (uchar_t *)sd_inq,
14919                     sizeof (struct scsi_inquiry), NULL, 0, inq83,
14920                     (size_t)inq83_len, &devid)) == DDI_SUCCESS) {
14921                         /*
14922                          * extract GUID from DEVID
14923                          */
14924                         guid = ddi_devid_to_guid(devid);

14926                         /*
14927                          * Do not enable MPXIO if the strlen(guid) is greater
14928                          * than MPTSAS_MAX_GUID_LEN, this constrain would be
14929                          * handled by framework later.
14930                          */
14931                         if (guid && (strlen(guid) > MPTSAS_MAX_GUID_LEN)) {
14932                                 ddi_devid_free_guid(guid);
14933                                 guid = NULL;
14934                                 if (mpt->m_mpxio_enable == TRUE) {
14935                                         mptsas_log(mpt, CE_NOTE, "!Target:%x, "
14936                                             "lun:%x doesn't have a valid GUID, "
14937                                             "multipathing for this drive is "
14938                                             "not enabled", target, lun);
14939                                 }
14940                         }

14942                         /*
14943                          * devid no longer needed
14944                          */
14945                         ddi_devid_free(devid);
14946                         break;
14947                 } else if (rval == DDI_NOT_WELL_FORMED) {
14948                         /*
14949                          * return value of ddi_devid_scsi_encode equal to
14950                          * DDI_NOT_WELL_FORMED means DEVID_RETRY, it worth
14951                          * to retry inquiry page 0x83 and get GUID.
```

```
14952                          */
14953                         NDBG20(("Not well formed devid, retry..."));
14954                         delay(1 * drv_usectohz(1000000));
14955                         continue;
14956                 } else {
14957                         mptsas_log(mpt, CE_WARN, "!Encode devid failed for "
14958                             "path target:%x, lun:%x", target, lun);
14959                         rval = DDI_FAILURE;
14960                         goto create_lun;
14961                 }
14962         }

14964         if (i == mptsas_inq83_retry_timeout) {
14965                 mptsas_log(mpt, CE_WARN, "!Repeated page83 requests timeout "
14966                     "for path target:%x, lun:%x", target, lun);
14967         }

14969         rval = DDI_FAILURE;

14971 create_lun:
14972         if ((guid != NULL) && (mpt->m_mpxio_enable == TRUE)) {
14973                 rval = mptsas_create_virt_lun(pdip, sd_inq, guid, lun_dip, &pip,
14974                     ptgt, lun);
14975         }
14976         if (rval != DDI_SUCCESS) {
14977                 rval = mptsas_create_phys_lun(pdip, sd_inq, guid, lun_dip,
14978                     ptgt, lun);

14980         }
14981 out:
14982         if (guid != NULL) {
14983                 /*
14984                  * guid no longer needed
14985                  */
14986                 ddi_devid_free_guid(guid);
14987         }
14988         if (inq83 != NULL)
14989                 kmem_free(inq83, inq83_len1);
14990         return (rval);
14991 }

14993 static int
14994 mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *inq, char *guid,
14995     dev_info_t **lun_dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt, int lun)
14996 {
14997         int                     target;
14998         char                    *nodename = NULL;
14999         char                    **compatible = NULL;
15000         int                     ncompatible     = 0;
15001         int                     mdi_rtn = MDI_FAILURE;
15002         int                     rval = DDI_FAILURE;
15003         char                    *old_guid = NULL;
15004         mptsas_t                *mpt = DIP2MPT(pdip);
15005         char                    *lun_addr = NULL;
15006         char                    *wwn_str = NULL;
15007         char                    *attached_wwn_str = NULL;
15008         char                    *component = NULL;
15009         uint8_t                 phy = 0xFF;
15010         uint64_t                sas_wwn;
15011         int64_t                 lun64 = 0;
15012         uint32_t                devinfo;
15013         uint16_t                dev_hdl;
15014         uint16_t                pdev_hdl;
15015         uint64_t                dev_sas_wwn;
15016         uint64_t                pdev_sas_wwn;
15017         uint32_t                pdev_info;
```



```
15018         uint8_t                 physport;
15019         uint8_t                 phy_id;
15020         uint32_t                page_address;
15021         uint16_t                bay_num, enclosure, io_flags;
15022         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
15023         uint32_t                dev_info;

15025         mutex_enter(&mpt->m_mutex);
15026         target = ptgt->m_devhdl;
15027         sas_wwn = ptgt->m_addr.mta_wwn;
15028         devinfo = ptgt->m_deviceinfo;
15029         phy = ptgt->m_phynum;
15030         mutex_exit(&mpt->m_mutex);

15032         if (sas_wwn) {
15033                 *pip = mptsas_find_path_addr(pdip, sas_wwn, lun);
15034         } else {
15035                 *pip = mptsas_find_path_phy(pdip, phy);
15036         }

15038         if (*pip != NULL) {
15039                 *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
15040                 ASSERT(*lun_dip != NULL);
15041                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, *lun_dip,
15042                     (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM),
15043                     MDI_CLIENT_GUID_PROP, &old_guid) == DDI_SUCCESS) {
15044                         if (strncmp(guid, old_guid, strlen(guid)) == 0) {
15045                                 /*
15046                                  * Same path back online again.
15047                                  */
15048                                 (void) ddi_prop_free(old_guid);
15049                                 if ((!MDI_PI_IS_ONLINE(*pip)) &&
15050                                     (!MDI_PI_IS_STANDBY(*pip)) &&
15051                                     (ptgt->m_tgt_unconfigured == 0)) {
15052                                         rval = mdi_pi_online(*pip, 0);
15053                                         mutex_enter(&mpt->m_mutex);
15054                                         ptgt->m_led_status = 0;
15055                                         (void) mptsas_flush_led_status(mpt,
15056                                             ptgt);
15057                                         mutex_exit(&mpt->m_mutex);
15058                                 } else {
15059                                         rval = DDI_SUCCESS;
15060                                 }
15061                                 if (rval != DDI_SUCCESS) {
15062                                         mptsas_log(mpt, CE_WARN, "path:target: "
15063                                             "%x, lun:%x online failed!", target,
15064                                             lun);
15065                                         *pip = NULL;
15066                                         *lun_dip = NULL;
15067                                 }
15068                                 return (rval);
15069                         } else {
15070                                 /*
15071                                  * The GUID of the LUN has changed which maybe
15072                                  * because customer mapped another volume to the
15073                                  * same LUN.
15074                                  */
15075                                 mptsas_log(mpt, CE_WARN, "The GUID of the "
15076                                     "target:%x, lun:%x was changed, maybe "
15077                                     "because someone mapped another volume "
15078                                     "to the same LUN", target, lun);
15079                                 (void) ddi_prop_free(old_guid);
15080                                 if (!MDI_PI_IS_OFFLINE(*pip)) {
15081                                         rval = mdi_pi_offline(*pip, 0);
15082                                         if (rval != MDI_SUCCESS) {
15083                                                 mptsas_log(mpt, CE_WARN, "path:"
```

```
15084                                                   "target:%x, lun:%x offline "
15085                                                   "failed!", target, lun);
15086                                           *pip = NULL;
15087                                           *lun_dip = NULL;
15088                                           return (DDI_FAILURE);
15089                                   }
15090                           }
15091                           if (mdi_pi_free(*pip, 0) != MDI_SUCCESS) {
15092                                   mptsas_log(mpt, CE_WARN, "path:target:"
15093                                       "%x, lun:%x free failed!", target,
15094                                       lun);
15095                                   *pip = NULL;
15096                                   *lun_dip = NULL;
15097                                   return (DDI_FAILURE);
15098                           }
15099                   }
15100           } else {
15101                   mptsas_log(mpt, CE_WARN, "Can't get client-guid "
15102                       "property for path:target:%x, lun:%x", target, lun);
15103                   *pip = NULL;
15104                   *lun_dip = NULL;
15105                   return (DDI_FAILURE);
15106           }
15107   }
15108   scsi_hba_nodename_compatible_get(inq, NULL,
15109       inq->inq_dtype, NULL, &nodename, &compatible, &ncompatible);

15111   /*
15112    * if nodename can't be determined then print a message and skip it
15113    */
15114   if (nodename == NULL) {
15115           mptsas_log(mpt, CE_WARN, "mptsas driver found no compatible "
15116               "driver for target%d lun %d dtype:0x%02x", target, lun,
15117               inq->inq_dtype);
15118           return (DDI_FAILURE);
15119   }

15121   wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
15122   /* The property is needed by MPAPI */
15123   (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);

15125   lun_addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
15126   if (guid) {
15127           (void) sprintf(lun_addr, "w%s,%x", wwn_str, lun);
15128           (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15129   } else {
15130           (void) sprintf(lun_addr, "p%x,%x", phy, lun);
15131           (void) sprintf(wwn_str, "p%x", phy);
15132   }

15134   mdi_rtn = mdi_pi_alloc_compatible(pdip, nodename,
15135       guid, lun_addr, compatible, ncompatible,
15136       0, pip);
15137   if (mdi_rtn == MDI_SUCCESS) {

15139           if (mdi_prop_update_string(*pip, MDI_GUID,
15140               guid) != DDI_SUCCESS) {
15141                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15142                       "create prop for target %d lun %d (MDI_GUID)",
15143                       target, lun);
15144                   mdi_rtn = MDI_FAILURE;
15145                   goto virt_create_done;
15146           }

15148           if (mdi_prop_update_int(*pip, LUN_PROP,
15149               lun) != DDI_SUCCESS) {
```



```
15150                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15151                       "create prop for target %d lun %d (LUN_PROP)",
15152                       target, lun);
15153                   mdi_rtn = MDI_FAILURE;
15154                   goto virt_create_done;
15155           }
15156           lun64 = (int64_t)lun;
15157           if (mdi_prop_update_int64(*pip, LUN64_PROP,
15158               lun64) != DDI_SUCCESS) {
15159                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15160                       "create prop for target %d (LUN64_PROP)",
15161                       target);
15162                   mdi_rtn = MDI_FAILURE;
15163                   goto virt_create_done;
15164           }
15165           if (mdi_prop_update_string_array(*pip, "compatible",
15166               compatible, ncompatible) !=
15167               DDI_PROP_SUCCESS) {
15168                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15169                       "create prop for target %d lun %d (COMPATIBLE)",
15170                       target, lun);
15171                   mdi_rtn = MDI_FAILURE;
15172                   goto virt_create_done;
15173           }
15174           if (sas_wwn && (mdi_prop_update_string(*pip,
15175               SCSI_ADDR_PROP_TARGET_PORT, wwn_str) != DDI_PROP_SUCCESS)) {
15176                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15177                       "create prop for target %d lun %d "
15178                       "(target-port)", target, lun);
15179                   mdi_rtn = MDI_FAILURE;
15180                   goto virt_create_done;
15181           } else if ((sas_wwn == 0) && (mdi_prop_update_int(*pip,
15182               "sata-phy", phy) != DDI_PROP_SUCCESS)) {
15183                   /*
15184                    * Direct attached SATA device without DeviceName
15185                    */
15186                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15187                       "create prop for SAS target %d lun %d "
15188                       "(sata-phy)", target, lun);
15189                   mdi_rtn = MDI_FAILURE;
15190                   goto virt_create_done;
15191           }
15192           mutex_enter(&mpt->m_mutex);

15194           page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15195               MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15196               (uint32_t)ptgt->m_devhdl;
15197           rval = mptsas_get_sas_device_page0(mpt, page_address,
15198               &dev_hdl, &dev_sas_wwn, &dev_info, &physport,
15199               &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15200           if (rval != DDI_SUCCESS) {
15201                   mutex_exit(&mpt->m_mutex);
15202                   mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15203                       "parent device for handle %d", page_address);
15204                   mdi_rtn = MDI_FAILURE;
15205                   goto virt_create_done;
15206           }

15208           page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15209               MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
15210           rval = mptsas_get_sas_device_page0(mpt, page_address,
15211               &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
15212               &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15213           if (rval != DDI_SUCCESS) {
15214                   mutex_exit(&mpt->m_mutex);
15215                   mptsas_log(mpt, CE_WARN, "mptsas unable to get"
```

```
15216                           "device info for handle %d", page_address);
15217                   mdi_rtn = MDI_FAILURE;
15218                   goto virt_create_done;
15219           }

15221           mutex_exit(&mpt->m_mutex);

15223           /*
15224            * If this device direct attached to the controller
15225            * set the attached-port to the base wwid
15226            */
15227           if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15228               != DEVINFO_DIRECT_ATTACHED) {
15229               (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15230                   pdev_sas_wwn);
15231           } else {
15232                   /*
15233                    * Update the iport's attached-port to guid
15234                    */
15235                   if (sas_wwn == 0) {
15236                           (void) sprintf(wwn_str, "p%x", phy);
15237                   } else {
15238                           (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15239                   }
15240                   if (ddi_prop_update_string(DDI_DEV_T_NONE,
15241                       pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
15242                       DDI_PROP_SUCCESS) {
15243                           mptsas_log(mpt, CE_WARN,
15244                               "mptsas unable to create "
15245                               "property for iport target-port"
15246                               " %s (sas_wwn)",
15247                               wwn_str);
15248                           mdi_rtn = MDI_FAILURE;
15249                           goto virt_create_done;
15250                   }

15252                   (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15253                       mpt->un.m_base_wwid);
15254           }

15256           if (mdi_prop_update_string(*pip,
15257               SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
15258               DDI_PROP_SUCCESS) {
15259                   mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15260                       "property for iport attached-port %s (sas_wwn)",
15261                       attached_wwn_str);
15262                   mdi_rtn = MDI_FAILURE;
15263                   goto virt_create_done;
15264           }


15267           if (inq->inq_dtype == 0) {
15268                   component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
15269                   /*
15270                    * set obp path for pathinfo
15271                    */
15272                   (void) snprintf(component, MAXPATHLEN,
15273                       "disk@%s", lun_addr);

15275                   if (mdi_pi_pathname_obp_set(*pip, component) !=
15276                       DDI_SUCCESS) {
15277                           mptsas_log(mpt, CE_WARN, "mpt_sas driver "
15278                               "unable to set obp-path for object %s",
15279                               component);
15280                           mdi_rtn = MDI_FAILURE;
15281                           goto virt_create_done;
```



```
15282                   }
15283           }

15285           *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
15286           if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
15287               MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
15288                   if ((ndi_prop_update_int(DDI_DEV_T_NONE, *lun_dip,
15289                       "pm-capable", 1)) !=
15290                       DDI_PROP_SUCCESS) {
15291                           mptsas_log(mpt, CE_WARN, "mptsas driver"
15292                               "failed to create pm-capable "
15293                               "property, target %d", target);
15294                           mdi_rtn = MDI_FAILURE;
15295                           goto virt_create_done;
15296                   }
15297           }
15298           /*
15299            * Create the phy-num property
15300            */
15301           if (mdi_prop_update_int(*pip, "phy-num",
15302               ptgt->m_phynum) != DDI_SUCCESS) {
15303                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15304                       "create phy-num property for target %d lun %d",
15305                       target, lun);
15306                   mdi_rtn = MDI_FAILURE;
15307                   goto virt_create_done;
15308           }
15309           NDBG20(("new path:%s onlining,", MDI_PI(*pip)->pi_addr));
15310           mdi_rtn = mdi_pi_online(*pip, 0);
15311           if (mdi_rtn == MDI_SUCCESS) {
15312                   mutex_enter(&mpt->m_mutex);
15313                   ptgt->m_led_status = 0;
15314                   (void) mptsas_flush_led_status(mpt, ptgt);
15315                   mutex_exit(&mpt->m_mutex);
15316           }
15317           if (mdi_rtn == MDI_NOT_SUPPORTED) {
15318                   mdi_rtn = MDI_FAILURE;
15319           }
15320   virt_create_done:
15321           if (*pip && mdi_rtn != MDI_SUCCESS) {
15322                   (void) mdi_pi_free(*pip, 0);
15323                   *pip = NULL;
15324                   *lun_dip = NULL;
15325           }
15326   }

15328   scsi_hba_nodename_compatible_free(nodename, compatible);
15329   if (lun_addr != NULL) {
15330           kmem_free(lun_addr, SCSI_MAXNAMELEN);
15331   }
15332   if (wwn_str != NULL) {
15333           kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
15334   }
15335   if (component != NULL) {
15336           kmem_free(component, MAXPATHLEN);
15337   }

15339   return ((mdi_rtn == MDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
15340 }

15342 static int
15343 mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *inq,
15344     char *guid, dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
15345 {
15346   int                     target;
15347   int                     rval;
```

```
15348         int                     ndi_rtn = NDI_FAILURE;
15349         uint64_t                be_sas_wwn;
15350         char                    *nodename = NULL;
15351         char                    **compatible = NULL;
15352         int                     ncompatible = 0;
15353         int                     instance = 0;
15354         mptsas_t                *mpt = DIP2MPT(pdip);
15355         char                    *wwn_str = NULL;
15356         char                    *component = NULL;
15357         char                    *attached_wwn_str = NULL;
15358         uint8_t                 phy = 0xFF;
15359         uint64_t                sas_wwn;
15360         uint32_t                devinfo;
15361         uint16_t                dev_hdl;
15362         uint16_t                pdev_hdl;
15363         uint64_t                pdev_sas_wwn;
15364         uint64_t                dev_sas_wwn;
15365         uint32_t                pdev_info;
15366         uint8_t                 physport;
15367         uint8_t                 phy_id;
15368         uint32_t                page_address;
15369         uint16_t                bay_num, enclosure, io_flags;
15370         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
15371         uint32_t                dev_info;
15372         int64_t                 lun64 = 0;

15374         mutex_enter(&mpt->m_mutex);
15375         target = ptgt->m_devhdl;
15376         sas_wwn = ptgt->m_addr.mta_wwn;
15377         devinfo = ptgt->m_deviceinfo;
15378         phy = ptgt->m_phynum;
15379         mutex_exit(&mpt->m_mutex);

15381         /*
15382          * generate compatible property with binding-set "mpt"
15383          */
15384         scsi_hba_nodename_compatible_get(inq, NULL, inq->inq_dtype, NULL,
15385             &nodename, &compatible, &ncompatible);

15387         /*
15388          * if nodename can't be determined then print a message and skip it
15389          */
15390         if (nodename == NULL) {
15391                 mptsas_log(mpt, CE_WARN, "mptsas found no compatible driver "
15392                     "for target %d lun %d", target, lun);
15393                 return (DDI_FAILURE);
15394         }

15396         ndi_rtn = ndi_devi_alloc(pdip, nodename,
15397             DEVI_SID_NODEID, lun_dip);

15399         /*
15400          * if lun alloc success, set props
15401          */
15402         if (ndi_rtn == NDI_SUCCESS) {

15404                 if (ndi_prop_update_int(DDI_DEV_T_NONE,
15405                     *lun_dip, LUN_PROP, lun) !=
15406                     DDI_PROP_SUCCESS) {
15407                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15408                             "property for target %d lun %d (LUN_PROP)",
15409                             target, lun);
15410                         ndi_rtn = NDI_FAILURE;
15411                         goto phys_create_done;
15412                 }
```



```
15414                 lun64 = (int64_t)lun;
15415                 if (ndi_prop_update_int64(DDI_DEV_T_NONE,
15416                     *lun_dip, LUN64_PROP, lun64) !=
15417                     DDI_PROP_SUCCESS) {
15418                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15419                             "property for target %d lun64 %d (LUN64_PROP)",
15420                             target, lun);
15421                         ndi_rtn = NDI_FAILURE;
15422                         goto phys_create_done;
15423                 }
15424                 if (ndi_prop_update_string_array(DDI_DEV_T_NONE,
15425                     *lun_dip, "compatible", compatible, ncompatible)
15426                     != DDI_PROP_SUCCESS) {
15427                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15428                             "property for target %d lun %d (COMPATIBLE)",
15429                             target, lun);
15430                         ndi_rtn = NDI_FAILURE;
15431                         goto phys_create_done;
15432                 }

15434                 /*
15435                  * We need the SAS WWN for non-multipath devices, so
15436                  * we'll use the same property as that multipathing
15437                  * devices need to present for MPAPI. If we don't have
15438                  * a WWN (e.g. parallel SCSI), don't create the prop.
15439                  */
15440                 wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
15441                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15442                 if (sas_wwn && ndi_prop_update_string(DDI_DEV_T_NONE,
15443                     *lun_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str)
15444                     != DDI_PROP_SUCCESS) {
15445                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15446                             "create property for SAS target %d lun %d "
15447                             "(target-port)", target, lun);
15448                         ndi_rtn = NDI_FAILURE;
15449                         goto phys_create_done;
15450                 }

15452                 be_sas_wwn = BE_64(sas_wwn);
15453                 if (sas_wwn && ndi_prop_update_byte_array(
15454                     DDI_DEV_T_NONE, *lun_dip, "port-wwn",
15455                     (uchar_t *)&be_sas_wwn, 8) != DDI_PROP_SUCCESS) {
15456                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15457                             "create property for SAS target %d lun %d "
15458                             "(port-wwn)", target, lun);
15459                         ndi_rtn = NDI_FAILURE;
15460                         goto phys_create_done;
15461                 } else if ((sas_wwn == 0) && (ndi_prop_update_int(
15462                     DDI_DEV_T_NONE, *lun_dip, "sata-phy", phy) !=
15463                     DDI_PROP_SUCCESS)) {
15464                         /*
15465                          * Direct attached SATA device without DeviceName
15466                          */
15467                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15468                             "create property for SAS target %d lun %d "
15469                             "(sata-phy)", target, lun);
15470                         ndi_rtn = NDI_FAILURE;
15471                         goto phys_create_done;
15472                 }

15474                 if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
15475                     *lun_dip, SAS_PROP) != DDI_PROP_SUCCESS) {
15476                         mptsas_log(mpt, CE_WARN, "mptsas unable to"
15477                             "create property for SAS target %d lun %d"
15478                             " (SAS_PROP)", target, lun);
15479                         ndi_rtn = NDI_FAILURE;
```

```
15480                           goto phys_create_done;
15481                   }
15482                   if (guid && (ndi_prop_update_string(DDI_DEV_T_NONE,
15483                       *lun_dip, NDI_GUID, guid) != DDI_SUCCESS)) {
15484                           mptsas_log(mpt, CE_WARN, "mptsas unable "
15485                               "to create guid property for target %d "
15486                               "lun %d", target, lun);
15487                           ndi_rtn = NDI_FAILURE;
15488                           goto phys_create_done;
15489                   }

15491                   /*
15492                    * The following code is to set properties for SM-HBA support,
15493                    * it doesn't apply to RAID volumes
15494                    */
15495                   if (ptgt->m_addr.mta_phymask == 0)
15496                           goto phys_raid_lun;

15498                   mutex_enter(&mpt->m_mutex);

15500                   page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15501                       MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15502                       (uint32_t)ptgt->m_devhdl;
15503                   rval = mptsas_get_sas_device_page0(mpt, page_address,
15504                       &dev_hdl, &dev_sas_wwn, &dev_info,
15505                       &physport, &phy_id, &pdev_hdl,
15506                       &bay_num, &enclosure, &io_flags);
15507                   if (rval != DDI_SUCCESS) {
15508                           mutex_exit(&mpt->m_mutex);
15509                           mptsas_log(mpt, CE_WARN, "mptsas unable to get"
15510                               "parent device for handle %d.", page_address);
15511                           ndi_rtn = NDI_FAILURE;
15512                           goto phys_create_done;
15513                   }

15515                   page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15516                       MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
15517                   rval = mptsas_get_sas_device_page0(mpt, page_address,
15518                       &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
15519                       &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15520                   if (rval != DDI_SUCCESS) {
15521                           mutex_exit(&mpt->m_mutex);
15522                           mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15523                               "device for handle %d.", page_address);
15524                           ndi_rtn = NDI_FAILURE;
15525                           goto phys_create_done;
15526                   }

15528                   mutex_exit(&mpt->m_mutex);

15530                   /*
15531                    * If this device direct attached to the controller
15532                    * set the attached-port to the base wwid
15533                    */
15534                   if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15535                       != DEVINFO_DIRECT_ATTACHED) {
15536                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15537                               pdev_sas_wwn);
15538                   } else {
15539                           /*
15540                            * Update the iport's attached-port to guid
15541                            */
15542                           if (sas_wwn == 0) {
15543                                   (void) sprintf(wwn_str, "p%x", phy);
15544                           } else {
15545                                   (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
```



```
15546                           }
15547                           if (ddi_prop_update_string(DDI_DEV_T_NONE,
15548                               pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
15549                               DDI_PROP_SUCCESS) {
15550                                   mptsas_log(mpt, CE_WARN,
15551                                       "mptsas unable to create "
15552                                       "property for iport target-port"
15553                                       " %s (sas_wwn)",
15554                                       wwn_str);
15555                                   ndi_rtn = NDI_FAILURE;
15556                                   goto phys_create_done;
15557                           }

15559                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15560                               mpt->un.m_base_wwid);
15561                   }

15563                   if (ndi_prop_update_string(DDI_DEV_T_NONE,
15564                       *lun_dip, SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
15565                       DDI_PROP_SUCCESS) {
15566                           mptsas_log(mpt, CE_WARN,
15567                               "mptsas unable to create "
15568                               "property for iport attached-port %s (sas_wwn)",
15569                               attached_wwn_str);
15570                           ndi_rtn = NDI_FAILURE;
15571                           goto phys_create_done;
15572                   }

15574                   if (IS_SATA_DEVICE(dev_info)) {
15575                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
15576                               *lun_dip, MPTSAS_VARIANT, "sata") !=
15577                               DDI_PROP_SUCCESS) {
15578                                   mptsas_log(mpt, CE_WARN,
15579                                       "mptsas unable to create "
15580                                       "property for device variant ");
15581                                   ndi_rtn = NDI_FAILURE;
15582                                   goto phys_create_done;
15583                           }
15584                   }

15586                   if (IS_ATAPI_DEVICE(dev_info)) {
15587                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
15588                               *lun_dip, MPTSAS_VARIANT, "atapi") !=
15589                               DDI_PROP_SUCCESS) {
15590                                   mptsas_log(mpt, CE_WARN,
15591                                       "mptsas unable to create "
15592                                       "property for device variant ");
15593                                   ndi_rtn = NDI_FAILURE;
15594                                   goto phys_create_done;
15595                           }
15596                   }

15598           phys_raid_lun:
15599                   /*
15600                    * if this is a SAS controller, and the target is a SATA
15601                    * drive, set the 'pm-capable' property for sd and if on
15602                    * an OPL platform, also check if this is an ATAPI
15603                    * device.
15604                    */
15605                   instance = ddi_get_instance(mpt->m_dip);
15606                   if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
15607                       MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
15608                           NDBG2(("mptsas%d: creating pm-capable property, "
15609                               "target %d", instance, target));

15611                           if ((ndi_prop_update_int(DDI_DEV_T_NONE,
```

```
15612                           *lun_dip, "pm-capable", 1)) !=
15613                           DDI_PROP_SUCCESS) {
15614                               mptsas_log(mpt, CE_WARN, "mptsas "
15615                                   "failed to create pm-capable "
15616                                   "property, target %d", target);
15617                               ndi_rtn = NDI_FAILURE;
15618                               goto phys_create_done;
15619                       }

15621                   }

15623                   if ((inq->inq_dtype == 0) || (inq->inq_dtype == 5)) {
15624                           /*
15625                            * add 'obp-path' properties for devinfo
15626                            */
15627                           bzero(wwn_str, sizeof (wwn_str));
15628                           (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
15629                           component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
15630                           if (guid) {
15631                                   (void) snprintf(component, MAXPATHLEN,
15632                                       "disk@w%s,%x", wwn_str, lun);
15633                           } else {
15634                                   (void) snprintf(component, MAXPATHLEN,
15635                                       "disk@p%x,%x", phy, lun);
15636                           }
15637                           if (ddi_pathname_obp_set(*lun_dip, component)
15638                               != DDI_SUCCESS) {
15639                               mptsas_log(mpt, CE_WARN, "mpt_sas driver "
15640                                   "unable to set obp-path for SAS "
15641                                   "object %s", component);
15642                               ndi_rtn = NDI_FAILURE;
15643                               goto phys_create_done;
15644                           }
15645                   }
15646                   /*
15647                    * Create the phy-num property for non-raid disk
15648                    */
15649                   if (ptgt->m_addr.mta_phymask != 0) {
15650                           if (ndi_prop_update_int(DDI_DEV_T_NONE,
15651                               *lun_dip, "phy-num", ptgt->m_phynum) !=
15652                               DDI_PROP_SUCCESS) {
15653                               mptsas_log(mpt, CE_WARN, "mptsas driver "
15654                                   "failed to create phy-num property for "
15655                                   "target %d", target);
15656                               ndi_rtn = NDI_FAILURE;
15657                               goto phys_create_done;
15658                           }
15659                   }
15660 phys_create_done:
15661                   /*
15662                    * If props were setup ok, online the lun
15663                    */
15664                   if (ndi_rtn == NDI_SUCCESS) {
15665                           /*
15666                            * Try to online the new node
15667                            */
15668                           ndi_rtn = ndi_devi_online(*lun_dip, NDI_ONLINE_ATTACH);
15669                   }
15670                   if (ndi_rtn == NDI_SUCCESS) {
15671                           mutex_enter(&mpt->m_mutex);
15672                           ptgt->m_led_status = 0;
15673                           (void) mptsas_flush_led_status(mpt, ptgt);
15674                           mutex_exit(&mpt->m_mutex);
15675                   }

15677                   /*
```



```
15678                    * If success set rtn flag, else unwire alloc'd lun
15679                    */
15680                   if (ndi_rtn != NDI_SUCCESS) {
15681                           NDBG12(("mptsas driver unable to online "
15682                               "target %d lun %d", target, lun));
15683                           ndi_prop_remove_all(*lun_dip);
15684                           (void) ndi_devi_free(*lun_dip);
15685                           *lun_dip = NULL;
15686                   }
15687           }

15689           scsi_hba_nodename_compatible_free(nodename, compatible);

15691           if (wwn_str != NULL) {
15692                   kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
15693           }
15694           if (component != NULL) {
15695                   kmem_free(component, MAXPATHLEN);
15696           }


15699           return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
15700 }

15702 static int
15703 mptsas_probe_smp(dev_info_t *pdip, uint64_t wwn)
15704 {
15705           mptsas_t        *mpt = DIP2MPT(pdip);
15706           struct smp_device smp_sd;

15708           /* XXX An HBA driver should not be allocating an smp_device. */
15709           bzero(&smp_sd, sizeof (struct smp_device));
15710           smp_sd.smp_sd_address.smp_a_hba_tran = mpt->m_smptran;
15711           bcopy(&wwn, smp_sd.smp_sd_address.smp_a_wwn, SAS_WWN_BYTE_SIZE);

15713           if (smp_probe(&smp_sd) != DDI_PROBE_SUCCESS)
15714                   return (NDI_FAILURE);
15715           return (NDI_SUCCESS);
15716 }

15718 static int
15719 mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn, dev_info_t **smp_dip)
15720 {
15721           mptsas_t        *mpt = DIP2MPT(pdip);
15722           mptsas_smp_t    *psmp = NULL;
15723           int             rval;
15724           int             phymask;

15726           /*
15727            * Get the physical port associated to the iport
15728            * PHYMASK TODO
15729            */
15730           phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
15731                   "phymask", 0);
15732           /*
15733            * Find the smp node in hash table with specified sas address and
15734            * physical port
15735            */
15736           psmp = mptsas_wwid_to_psmp(mpt, phymask, sas_wwn);
15737           if (psmp == NULL) {
15738                   return (DDI_FAILURE);
15739           }

15741           rval = mptsas_online_smp(pdip, psmp, smp_dip);

15743           return (rval);
```

```
15744 }

15746 static int
15747 mptsas_online_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
15748     dev_info_t **smp_dip)
15749 {
15750         char            wwn_str[MPTSAS_WWN_STRLEN];
15751         char            attached_wwn_str[MPTSAS_WWN_STRLEN];
15752         int             ndi_rtn = NDI_FAILURE;
15753         int             rval = 0;
15754         mptsas_smp_t    dev_info;
15755         uint32_t        page_address;
15756         mptsas_t        *mpt = DIP2MPT(pdip);
15757         uint16_t        dev_hdl;
15758         uint64_t        sas_wwn;
15759         uint64_t        smp_sas_wwn;
15760         uint8_t         physport;
15761         uint8_t         phy_id;
15762         uint16_t        pdev_hdl;
15763         uint8_t         numphys = 0;
15764         uint16_t        i = 0;
15765         char            phymask[MPTSAS_MAX_PHYS];
15766         char            *iport = NULL;
15767         mptsas_phymask_t        phy_mask = 0;
15768         uint16_t        attached_devhdl;
15769         uint16_t        bay_num, enclosure, io_flags;

15771         (void) sprintf(wwn_str, "%"PRIx64, smp_node->m_addr.mta_wwn);

15773         /*
15774          * Probe smp device, prevent the node of removed device from being
15775          * configured succesfully
15776          */
15777         if (mptsas_probe_smp(pdip, smp_node->m_addr.mta_wwn) != NDI_SUCCESS) {
15778                 return (DDI_FAILURE);
15779         }

15781         if ((*smp_dip = mptsas_find_smp_child(pdip, wwn_str)) != NULL) {
15782                 return (DDI_SUCCESS);
15783         }

15785         ndi_rtn = ndi_devi_alloc(pdip, "smp", DEVI_SID_NODEID, smp_dip);

15787         /*
15788          * if lun alloc success, set props
15789          */
15790         if (ndi_rtn == NDI_SUCCESS) {
15791                 /*
15792                  * Set the flavor of the child to be SMP flavored
15793                  */
15794                 ndi_flavor_set(*smp_dip, SCSA_FLAVOR_SMP);

15796                 if (ndi_prop_update_string(DDI_DEV_T_NONE,
15797                     *smp_dip, SMP_WWN, wwn_str) !=
15798                     DDI_PROP_SUCCESS) {
15799                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15800                             "property for smp device %s (sas_wwn)",
15801                             wwn_str);
15802                         ndi_rtn = NDI_FAILURE;
15803                         goto smp_create_done;
15804                 }
15805                 (void) sprintf(wwn_str, "w%"PRIx64, smp_node->m_addr.mta_wwn);
15806                 if (ndi_prop_update_string(DDI_DEV_T_NONE,
15807                     *smp_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str) !=
15808                     DDI_PROP_SUCCESS) {
15809                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
```



```
15810                             "property for iport target-port %s (sas_wwn)",
15811                             wwn_str);
15812                         ndi_rtn = NDI_FAILURE;
15813                         goto smp_create_done;
15814                 }

15816                 mutex_enter(&mpt->m_mutex);

15818                 page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
15819                     MPI2_SAS_EXPAND_PGAD_FORM_MASK) | smp_node->m_devhdl;
15820                 rval = mptsas_get_sas_expander_page0(mpt, page_address,
15821                     &dev_info);
15822                 if (rval != DDI_SUCCESS) {
15823                         mutex_exit(&mpt->m_mutex);
15824                         mptsas_log(mpt, CE_WARN,
15825                             "mptsas unable to get expander "
15826                             "parent device info for %x", page_address);
15827                         ndi_rtn = NDI_FAILURE;
15828                         goto smp_create_done;
15829                 }

15831                 smp_node->m_pdevhdl = dev_info.m_pdevhdl;
15832                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15833                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15834                     (uint32_t)dev_info.m_pdevhdl;
15835                 rval = mptsas_get_sas_device_page0(mpt, page_address,
15836                     &dev_hdl, &sas_wwn, &smp_node->m_pdevinfo, &physport,
15837                     &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15838                 if (rval != DDI_SUCCESS) {
15839                         mutex_exit(&mpt->m_mutex);
15840                         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15841                             "device info for %x", page_address);
15842                         ndi_rtn = NDI_FAILURE;
15843                         goto smp_create_done;
15844                 }

15846                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15847                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15848                     (uint32_t)dev_info.m_devhdl;
15849                 rval = mptsas_get_sas_device_page0(mpt, page_address,
15850                     &dev_hdl, &smp_sas_wwn, &smp_node->m_deviceinfo,
15851                     &physport, &phy_id, &pdev_hdl, &bay_num, &enclosure,
15852                     &io_flags);
15853                 if (rval != DDI_SUCCESS) {
15854                         mutex_exit(&mpt->m_mutex);
15855                         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15856                             "device info for %x", page_address);
15857                         ndi_rtn = NDI_FAILURE;
15858                         goto smp_create_done;
15859                 }
15860                 mutex_exit(&mpt->m_mutex);

15862                 /*
15863                  * If this smp direct attached to the controller
15864                  * set the attached-port to the base wwid
15865                  */
15866                 if ((smp_node->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15867                     != DEVINFO_DIRECT_ATTACHED) {
15868                         (void) sprintf(attached_wwn_str, "w%016"PRIx64,
15869                             sas_wwn);
15870                 } else {
15871                         (void) sprintf(attached_wwn_str, "w%016"PRIx64,
15872                             mpt->un.m_base_wwid);
15873                 }

15875                 if (ndi_prop_update_string(DDI_DEV_T_NONE,
```

```
15876                     *smp_dip, SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwn_str) !=
15877                     DDI_PROP_SUCCESS) {
15878                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15879                             "property for smp attached-port %s (sas_wwn)",
15880                             attached_wwn_str);
15881                         ndi_rtn = NDI_FAILURE;
15882                         goto smp_create_done;
15883                 }

15885                 if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
15886                     *smp_dip, SMP_PROP) != DDI_PROP_SUCCESS) {
15887                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15888                             "create property for SMP %s (SMP_PROP) ",
15889                             wwn_str);
15890                         ndi_rtn = NDI_FAILURE;
15891                         goto smp_create_done;
15892                 }

15894                 /*
15895                  * check the smp to see whether it direct
15896                  * attached to the controller
15897                  */
15898                 if ((smp_node->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15899                     != DEVINFO_DIRECT_ATTACHED) {
15900                         goto smp_create_done;
15901                 }
15902                 numphys = ddi_prop_get_int(DDI_DEV_T_ANY, pdip,
15903                     DDI_PROP_DONTPASS, MPTSAS_NUM_PHYS, -1);
15904                 if (numphys > 0) {
15905                         goto smp_create_done;
15906                 }
15907                 /*
15908                  * this iport is an old iport, we need to
15909                  * reconfig the props for it.
15910                  */
15911                 if (ddi_prop_update_int(DDI_DEV_T_NONE, pdip,
15912                     MPTSAS_VIRTUAL_PORT, 0) !=
15913                     DDI_PROP_SUCCESS) {
15914                         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15915                             MPTSAS_VIRTUAL_PORT);
15916                         mptsas_log(mpt, CE_WARN, "mptsas virtual port "
15917                             "prop update failed");
15918                         goto smp_create_done;
15919                 }

15921                 mutex_enter(&mpt->m_mutex);
15922                 numphys = 0;
15923                 iport = ddi_get_name_addr(pdip);
15924                 for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
15925                         bzero(phymask, sizeof (phymask));
15926                         (void) sprintf(phymask,
15927                             "%x", mpt->m_phy_info[i].phy_mask);
15928                         if (strcmp(phymask, iport) == 0) {
15929                                 phy_mask = mpt->m_phy_info[i].phy_mask;
15930                                 break;
15931                         }
15932                 }

15934                 for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
15935                         if ((phy_mask >> i) & 0x01) {
15936                                 numphys++;
15937                         }
15938                 }
15939                 /*
15940                  * Update PHY info for smhba
15941                  */
```



```
15942                 if (mptsas_smhba_phy_init(mpt)) {
15943                         mutex_exit(&mpt->m_mutex);
15944                         mptsas_log(mpt, CE_WARN, "mptsas phy update "
15945                             "failed");
15946                         goto smp_create_done;
15947                 }
15948                 mutex_exit(&mpt->m_mutex);

15950                 mptsas_smhba_set_all_phy_props(mpt, pdip, numphys, phy_mask,
15951                     &attached_devhdl);

15953                 if (ddi_prop_update_int(DDI_DEV_T_NONE, pdip,
15954                     MPTSAS_NUM_PHYS, numphys) !=
15955                     DDI_PROP_SUCCESS) {
15956                         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15957                             MPTSAS_NUM_PHYS);
15958                         mptsas_log(mpt, CE_WARN, "mptsas update "
15959                             "num phys props failed");
15960                         goto smp_create_done;
15961                 }
15962                 /*
15963                  * Add parent's props for SMHBA support
15964                  */
15965                 if (ddi_prop_update_string(DDI_DEV_T_NONE, pdip,
15966                     SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
15967                     DDI_PROP_SUCCESS) {
15968                         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15969                             SCSI_ADDR_PROP_ATTACHED_PORT);
15970                         mptsas_log(mpt, CE_WARN, "mptsas update iport"
15971                             "attached-port failed");
15972                         goto smp_create_done;
15973                 }

15975 smp_create_done:
15976                 /*
15977                  * If props were setup ok, online the lun
15978                  */
15979                 if (ndi_rtn == NDI_SUCCESS) {
15980                         /*
15981                          * Try to online the new node
15982                          */
15983                         ndi_rtn = ndi_devi_online(*smp_dip, NDI_ONLINE_ATTACH);
15984                 }

15986                 /*
15987                  * If success set rtn flag, else unwire alloc'd lun
15988                  */
15989                 if (ndi_rtn != NDI_SUCCESS) {
15990                         NDBG12(("mptsas unable to online "
15991                             "SMP target %s", wwn_str));
15992                         ndi_prop_remove_all(*smp_dip);
15993                         (void) ndi_devi_free(*smp_dip);
15994                 }
15995         }

15997         return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
15998 }

16000 /* smp transport routine */
16001 static int mptsas_smp_start(struct smp_pkt *smp_pkt)
16002 {
16003         uint64_t                        wwn;
16004         Mpi2SmpPassthroughRequest_t     req;
16005         Mpi2SmpPassthroughReply_t       rep;
16006         uint32_t                        direction = 0;
16007         mptsas_t                        *mpt;
```

```
16008         int                             ret;
16009         uint64_t                        tmp64;

16011         mpt = (mptsas_t *)smp_pkt->smp_pkt_address->
16012             smp_a_hba_tran->smp_tran_hba_private;

16014         bcopy(smp_pkt->smp_pkt_address->smp_a_wwn, &wwn, SAS_WWN_BYTE_SIZE);
16015         /*
16016          * Need to compose a SMP request message
16017          * and call mptsas_do_passthru() function
16018          */
16019         bzero(&req, sizeof (req));
16020         bzero(&rep, sizeof (rep));
16021         req.PassthroughFlags = 0;
16022         req.PhysicalPort = 0xff;
16023         req.ChainOffset = 0;
16024         req.Function = MPI2_FUNCTION_SMP_PASSTHROUGH;

16026         if ((smp_pkt->smp_pkt_reqsize & 0xffff0000ul) != 0) {
16027                 smp_pkt->smp_pkt_reason = ERANGE;
16028                 return (DDI_FAILURE);
16029         }
16030         req.RequestDataLength = LE_16((uint16_t)(smp_pkt->smp_pkt_reqsize - 4));

16032         req.MsgFlags = 0;
16033         tmp64 = LE_64(wwn);
16034         bcopy(&tmp64, &req.SASAddress, SAS_WWN_BYTE_SIZE);
16035         if (smp_pkt->smp_pkt_rspsize > 0) {
16036                 direction |= MPTSAS_PASS_THRU_DIRECTION_READ;
16037         }
16038         if (smp_pkt->smp_pkt_reqsize > 0) {
16039                 direction |= MPTSAS_PASS_THRU_DIRECTION_WRITE;
16040         }

16042         mutex_enter(&mpt->m_mutex);
16043         ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep,
16044             (uint8_t *)smp_pkt->smp_pkt_rsp,
16045             offsetof(Mpi2SmpPassthroughRequest_t, SGL), sizeof (rep),
16046             smp_pkt->smp_pkt_rspsize - 4, direction,
16047             (uint8_t *)smp_pkt->smp_pkt_req, smp_pkt->smp_pkt_reqsize - 4,
16048             smp_pkt->smp_pkt_timeout, FKIOCTL);
16049         mutex_exit(&mpt->m_mutex);
16050         if (ret != 0) {
16051                 cmn_err(CE_WARN, "smp_start do passthru error %d", ret);
16052                 smp_pkt->smp_pkt_reason = (uchar_t)(ret);
16053                 return (DDI_FAILURE);
16054         }
16055         /* do passthrough success, check the smp status */
16056         if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
16057                 switch (LE_16(rep.IOCStatus)) {
16058                 case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
16059                         smp_pkt->smp_pkt_reason = ENODEV;
16060                         break;
16061                 case MPI2_IOCSTATUS_SAS_SMP_DATA_OVERRUN:
16062                         smp_pkt->smp_pkt_reason = EOVERFLOW;
16063                         break;
16064                 case MPI2_IOCSTATUS_SAS_SMP_REQUEST_FAILED:
16065                         smp_pkt->smp_pkt_reason = EIO;
16066                         break;
16067                 default:
16068                         mptsas_log(mpt, CE_NOTE, "smp_start: get unknown ioc"
16069                             "status:%x", LE_16(rep.IOCStatus));
16070                         smp_pkt->smp_pkt_reason = EIO;
16071                         break;
16072                 }
16073                 return (DDI_FAILURE);
```



```
16074         }
16075         if (rep.SASStatus != MPI2_SASSTATUS_SUCCESS) {
16076                 mptsas_log(mpt, CE_NOTE, "smp_start: get error SAS status:%x",
16077                     rep.SASStatus);
16078                 smp_pkt->smp_pkt_reason = EIO;
16079                 return (DDI_FAILURE);
16080         }

16082         return (DDI_SUCCESS);
16083 }

16085 /*
16086  * If we didn't get a match, we need to get sas page0 for each device, and
16087  * untill we get a match. If failed, return NULL
16088  */
16089 static mptsas_target_t *
16090 mptsas_phy_to_tgt(mptsas_t *mpt, mptsas_phymask_t phymask, uint8_t phy)
16091 {
16092         int             i, j = 0;
16093         int             rval = 0;
16094         uint16_t        cur_handle;
16095         uint32_t        page_address;
16096         mptsas_target_t *ptgt = NULL;

16098         /*
16099          * PHY named device must be direct attached and attaches to
16100          * narrow port, if the iport is not parent of the device which
16101          * we are looking for.
16102          */
16103         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
16104                 if ((1 << i) & phymask)
16105                         j++;
16106         }

16108         if (j > 1)
16109                 return (NULL);

16111         /*
16112          * Must be a narrow port and single device attached to the narrow port
16113          * So the physical port num of device  which is equal to the iport's
16114          * port num is the device what we are looking for.
16115          */

16117         if (mpt->m_phy_info[phy].phy_mask != phymask)
16118                 return (NULL);

16120         mutex_enter(&mpt->m_mutex);

16122         ptgt = refhash_linear_search(mpt->m_targets, mptsas_target_eval_nowwn,
16123             &phy);
16124         if (ptgt != NULL) {
16125                 mutex_exit(&mpt->m_mutex);
16126                 return (ptgt);
16127         }

16129         if (mpt->m_done_traverse_dev) {
16130                 mutex_exit(&mpt->m_mutex);
16131                 return (NULL);
16132         }

16134         /* If didn't get a match, come here */
16135         cur_handle = mpt->m_dev_handle;
16136         for (; ; ) {
16137                 ptgt = NULL;
16138                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
16139                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)cur_handle;
```

```
16140                 rval = mptsas_get_target_device_info(mpt, page_address,
16141                     &cur_handle, &ptgt);
16142                 if ((rval == DEV_INFO_FAIL_PAGE0) ||
16143                     (rval == DEV_INFO_FAIL_ALLOC)) {
16144                         break;
16145                 }
16146                 if ((rval == DEV_INFO_WRONG_DEVICE_TYPE) ||
16147                     (rval == DEV_INFO_PHYS_DISK)) {
16148                         continue;
16149                 }
16150                 mpt->m_dev_handle = cur_handle;

16152                 if ((ptgt->m_addr.mta_wwn == 0) && (ptgt->m_phynum == phy)) {
16153                         break;
16154                 }
16155         }

16157         mutex_exit(&mpt->m_mutex);
16158         return (ptgt);
16159 }

16161 /*
16162  * The ptgt->m_addr.mta_wwn contains the wwid for each disk.
16163  * For Raid volumes, we need to check m_raidvol[x].m_raidwwid
16164  * If we didn't get a match, we need to get sas page0 for each device, and
16165  * untill we get a match
16166  * If failed, return NULL
16167  */
16168 static mptsas_target_t *
16169 mptsas_wwid_to_ptgt(mptsas_t *mpt, mptsas_phymask_t phymask, uint64_t wwid)
16170 {
16171         int             rval = 0;
16172         uint16_t        cur_handle;
16173         uint32_t        page_address;
16174         mptsas_target_t *tmp_tgt = NULL;
16175         mptsas_target_addr_t addr;

16177         addr.mta_wwn = wwid;
16178         addr.mta_phymask = phymask;
16179         mutex_enter(&mpt->m_mutex);
16180         tmp_tgt = refhash_lookup(mpt->m_targets, &addr);
16181         if (tmp_tgt != NULL) {
16182                 mutex_exit(&mpt->m_mutex);
16183                 return (tmp_tgt);
16184         }

16186         if (phymask == 0) {
16187                 /*
16188                  * It's IR volume
16189                  */
16190                 rval = mptsas_get_raid_info(mpt);
16191                 if (rval) {
16192                         tmp_tgt = refhash_lookup(mpt->m_targets, &addr);
16193                 }
16194                 mutex_exit(&mpt->m_mutex);
16195                 return (tmp_tgt);
16196         }

16198         if (mpt->m_done_traverse_dev) {
16199                 mutex_exit(&mpt->m_mutex);
16200                 return (NULL);
16201         }

16203         /* If didn't get a match, come here */
16204         cur_handle = mpt->m_dev_handle;
16205         for (;;) {
```



```
16206                 tmp_tgt = NULL;
16207                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
16208                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | cur_handle;
16209                 rval = mptsas_get_target_device_info(mpt, page_address,
16210                     &cur_handle, &tmp_tgt);
16211                 if ((rval == DEV_INFO_FAIL_PAGE0) ||
16212                     (rval == DEV_INFO_FAIL_ALLOC)) {
16213                         tmp_tgt = NULL;
16214                         break;
16215                 }
16216                 if ((rval == DEV_INFO_WRONG_DEVICE_TYPE) ||
16217                     (rval == DEV_INFO_PHYS_DISK)) {
16218                         continue;
16219                 }
16220                 mpt->m_dev_handle = cur_handle;
16221                 if ((tmp_tgt->m_addr.mta_wwn) &&
16222                     (tmp_tgt->m_addr.mta_wwn == wwid) &&
16223                     (tmp_tgt->m_addr.mta_phymask == phymask)) {
16224                         break;
16225                 }
16226         }

16228         mutex_exit(&mpt->m_mutex);
16229         return (tmp_tgt);
16230 }

16232 static mptsas_smp_t *
16233 mptsas_wwid_to_psmp(mptsas_t *mpt, mptsas_phymask_t phymask, uint64_t wwid)
16234 {
16235         int             rval = 0;
16236         uint16_t        cur_handle;
16237         uint32_t        page_address;
16238         mptsas_smp_t    smp_node, *psmp = NULL;
16239         mptsas_target_addr_t addr;

16241         addr.mta_wwn = wwid;
16242         addr.mta_phymask = phymask;
16243         mutex_enter(&mpt->m_mutex);
16244         psmp = refhash_lookup(mpt->m_smp_targets, &addr);
16245         if (psmp != NULL) {
16246                 mutex_exit(&mpt->m_mutex);
16247                 return (psmp);
16248         }

16250         if (mpt->m_done_traverse_smp) {
16251                 mutex_exit(&mpt->m_mutex);
16252                 return (NULL);
16253         }

16255         /* If didn't get a match, come here */
16256         cur_handle = mpt->m_smp_devhdl;
16257         for (;;) {
16258                 psmp = NULL;
16259                 page_address = (MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL &
16260                     MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)cur_handle;
16261                 rval = mptsas_get_sas_expander_page0(mpt, page_address,
16262                     &smp_node);
16263                 if (rval != DDI_SUCCESS) {
16264                         break;
16265                 }
16266                 mpt->m_smp_devhdl = cur_handle = smp_node.m_devhdl;
16267                 psmp = mptsas_smp_alloc(mpt, &smp_node);
16268                 ASSERT(psmp);
16269                 if ((psmp->m_addr.mta_wwn) && (psmp->m_addr.mta_wwn == wwid) &&
16270                     (psmp->m_addr.mta_phymask == phymask)) {
16271                         break;
```

```
16272                 }
16273         }

16275         mutex_exit(&mpt->m_mutex);
16276         return (psmp);
16277 }

16279 mptsas_target_t *
16280 mptsas_tgt_alloc(mptsas_t *mpt, uint16_t devhdl, uint64_t wwid,
16281     uint32_t devinfo, mptsas_phymask_t phymask, uint8_t phynum)
16282 {
16283         mptsas_target_t *tmp_tgt = NULL;
16284         mptsas_target_addr_t addr;

16286         addr.mta_wwn = wwid;
16287         addr.mta_phymask = phymask;
16288         tmp_tgt = refhash_lookup(mpt->m_targets, &addr);
16289         if (tmp_tgt != NULL) {
16290                 NDBG20(("Hash item already exist"));
16291                 tmp_tgt->m_deviceinfo = devinfo;
16292                 tmp_tgt->m_devhdl = devhdl;     /* XXX - duplicate? */
16293                 return (tmp_tgt);
16294         }
16295         tmp_tgt = kmem_zalloc(sizeof (struct mptsas_target), KM_SLEEP);
16296         if (tmp_tgt == NULL) {
16297                 cmn_err(CE_WARN, "Fatal, allocated tgt failed");
16298                 return (NULL);
16299         }
16300         tmp_tgt->m_devhdl = devhdl;
16301         tmp_tgt->m_addr.mta_wwn = wwid;
16302         tmp_tgt->m_deviceinfo = devinfo;
16303         tmp_tgt->m_addr.mta_phymask = phymask;
16304         tmp_tgt->m_phynum = phynum;
16305         /* Initialized the tgt structure */
16306         tmp_tgt->m_qfull_retries = QFULL_RETRIES;
16307         tmp_tgt->m_qfull_retry_interval =
16308             drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
16309         tmp_tgt->m_t_throttle = MAX_THROTTLE;
16310         TAILQ_INIT(&tmp_tgt->m_active_cmdq);

16312         refhash_insert(mpt->m_targets, tmp_tgt);

16314         return (tmp_tgt);
16315 }

16317 static void
16318 mptsas_smp_target_copy(mptsas_smp_t *src, mptsas_smp_t *dst)
16319 {
16320         dst->m_devhdl = src->m_devhdl;
16321         dst->m_deviceinfo = src->m_deviceinfo;
16322         dst->m_pdevhdl = src->m_pdevhdl;
16323         dst->m_pdevinfo = src->m_pdevinfo;
16324 }

16326 static mptsas_smp_t *
16327 mptsas_smp_alloc(mptsas_t *mpt, mptsas_smp_t *data)
16328 {
16329         mptsas_target_addr_t addr;
16330         mptsas_smp_t *ret_data;

16332         addr.mta_wwn = data->m_addr.mta_wwn;
16333         addr.mta_phymask = data->m_addr.mta_phymask;
16334         ret_data = refhash_lookup(mpt->m_smp_targets, &addr);
16335         /*
16336          * If there's already a matching SMP target, update its fields
16337          * in place.  Since the address is not changing, it's safe to do
```



```
16338          * this.  We cannot just bcopy() here because the structure we've
16339          * been given has invalid hash links.
16340          */
16341         if (ret_data != NULL) {
16342                 mptsas_smp_target_copy(data, ret_data);
16343                 return (ret_data);
16344         }

16346         ret_data = kmem_alloc(sizeof (mptsas_smp_t), KM_SLEEP);
16347         bcopy(data, ret_data, sizeof (mptsas_smp_t));
16348         refhash_insert(mpt->m_smp_targets, ret_data);
16349         return (ret_data);
16350 }

16352 /*
16353  * Functions for SGPIO LED support
16354  */
16355 static dev_info_t *
16356 mptsas_get_dip_from_dev(dev_t dev, mptsas_phymask_t *phymask)
16357 {
16358         dev_info_t      *dip;
16359         int             prop;
16360         dip = e_ddi_hold_devi_by_dev(dev, 0);
16361         if (dip == NULL)
16362                 return (dip);
16363         prop = ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0,
16364             "phymask", 0);
16365         *phymask = (mptsas_phymask_t)prop;
16366         ddi_release_devi(dip);
16367         return (dip);
16368 }
16369 static mptsas_target_t *
16370 mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr, mptsas_phymask_t phymask)
16371 {
16372         uint8_t                 phynum;
16373         uint64_t                wwn;
16374         int                     lun;
16375         mptsas_target_t         *ptgt = NULL;

16377         if (mptsas_parse_address(addr, &wwn, &phynum, &lun) != DDI_SUCCESS) {
16378                 return (NULL);
16379         }
16380         if (addr[0] == 'w') {
16381                 ptgt = mptsas_wwid_to_ptgt(mpt, (int)phymask, wwn);
16382         } else {
16383                 ptgt = mptsas_phy_to_tgt(mpt, (int)phymask, phynum);
16384         }
16385         return (ptgt);
16386 }

16388 static int
16389 mptsas_flush_led_status(mptsas_t *mpt, mptsas_target_t *ptgt)
16390 {
16391         uint32_t slotstatus = 0;

16393         /* Build an MPI2 Slot Status based on our view of the world */
16394         if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_IDENT - 1)))
16395                 slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_IDENTIFY_REQUEST;
16396         if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_FAIL - 1)))
16397                 slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_PREDICTED_FAULT;
16398         if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1)))
16399                 slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;

16401         /* Write it to the controller */
16402         NDBG14(("mptsas_ioctl: set LED status %x for slot %x",
16403             slotstatus, ptgt->m_slot_num));
```

```
16404         return (mptsas_send_sep(mpt, ptgt, &slotstatus,
16405             MPI2_SEP_REQ_ACTION_WRITE_STATUS));
16406 }

16408 /*
16409  *  send sep request, use enclosure/slot addressing
16410  */
16411 static int
16412 mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
16413     uint32_t *status, uint8_t act)
16414 {
16415         Mpi2SepRequest_t        req;
16416         Mpi2SepReply_t          rep;
16417         int                     ret;

16419         ASSERT(mutex_owned(&mpt->m_mutex));

16421         /*
16422          * We only support SEP control of directly-attached targets, in which
16423          * case the "SEP" we're talking to is a virtual one contained within
16424          * the HBA itself.  This is necessary because DA targets typically have
16425          * no other mechanism for LED control.  Targets for which a separate
16426          * enclosure service processor exists should be controlled via ses(7d)
16427          * or sgen(7d).  Furthermore, since such requests can time out, they
16428          * should be made in user context rather than in response to
16429          * asynchronous fabric changes.
16430          *
16431          * In addition, we do not support this operation for RAID volumes,
16432          * since there is no slot associated with them.
16433          */
16434         if (!(ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED) ||
16435             ptgt->m_addr.mta_phymask == 0) {
16436                 return (ENOTTY);
16437         }

16439         bzero(&req, sizeof (req));
16440         bzero(&rep, sizeof (rep));

16442         req.Function = MPI2_FUNCTION_SCSI_ENCLOSURE_PROCESSOR;
16443         req.Action = act;
16444         req.Flags = MPI2_SEP_REQ_FLAGS_ENCLOSURE_SLOT_ADDRESS;
16445         req.EnclosureHandle = LE_16(ptgt->m_enclosure);
16446         req.Slot = LE_16(ptgt->m_slot_num);
16447         if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16448                 req.SlotStatus = LE_32(*status);
16449         }
16450         ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
16451             sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
16452         if (ret != 0) {
16453                 mptsas_log(mpt, CE_NOTE, "mptsas_send_sep: passthru SEP "
16454                     "Processor Request message error %d", ret);
16455                 return (ret);
16456         }
16457         /* do passthrough success, check the ioc status */
16458         if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
16459                 mptsas_log(mpt, CE_NOTE, "send_sep act %x: ioc "
16460                     "status:%x loginfo %x", act, LE_16(rep.IOCStatus),
16461                     LE_32(rep.IOCLogInfo));
16462                 switch (LE_16(rep.IOCStatus) & MPI2_IOCSTATUS_MASK) {
16463                 case MPI2_IOCSTATUS_INVALID_FUNCTION:
16464                 case MPI2_IOCSTATUS_INVALID_VPID:
16465                 case MPI2_IOCSTATUS_INVALID_FIELD:
16466                 case MPI2_IOCSTATUS_INVALID_STATE:
16467                 case MPI2_IOCSTATUS_OP_STATE_NOT_SUPPORTED:
16468                 case MPI2_IOCSTATUS_CONFIG_INVALID_ACTION:
16469                 case MPI2_IOCSTATUS_CONFIG_INVALID_TYPE:
```



```
16470                 case MPI2_IOCSTATUS_CONFIG_INVALID_PAGE:
16471                 case MPI2_IOCSTATUS_CONFIG_INVALID_DATA:
16472                 case MPI2_IOCSTATUS_CONFIG_NO_DEFAULTS:
16473                         return (EINVAL);
16474                 case MPI2_IOCSTATUS_BUSY:
16475                         return (EBUSY);
16476                 case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
16477                         return (EAGAIN);
16478                 case MPI2_IOCSTATUS_INVALID_SGL:
16479                 case MPI2_IOCSTATUS_INTERNAL_ERROR:
16480                 case MPI2_IOCSTATUS_CONFIG_CANT_COMMIT:
16481                 default:
16482                         return (EIO);
16483                 }
16484         }
16485         if (act != MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16486                 *status = LE_32(rep.SlotStatus);
16487         }

16489         return (0);
16490 }

16492 int
16493 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
16494     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
16495     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
16496 {
16497         ddi_dma_cookie_t        new_cookie;
16498         size_t                  alloc_len;
16499         uint_t                  ncookie;

16501         if (cookiep == NULL)
16502                 cookiep = &new_cookie;

16504         if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
16505             NULL, dma_hdp) != DDI_SUCCESS) {
16506                 return (FALSE);
16507         }

16509         if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
16510             DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
16511             acc_hdp) != DDI_SUCCESS) {
16512                 ddi_dma_free_handle(dma_hdp);
16513                 *dma_hdp = NULL;
16514                 return (FALSE);
16515         }

16517         if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
16518             (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
16519             cookiep, &ncookie) != DDI_DMA_MAPPED) {
16520                 (void) ddi_dma_mem_free(acc_hdp);
16521                 ddi_dma_free_handle(dma_hdp);
16522                 *dma_hdp = NULL;
16523                 return (FALSE);
16524         }

16526         return (TRUE);
16527 }

16529 void
16530 mptsas_dma_addr_destroy(ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp)
16531 {
16532         if (*dma_hdp == NULL)
16533                 return;

16535         (void) ddi_dma_unbind_handle(*dma_hdp);
```

```
16536         (void) ddi_dma_mem_free(acc_hdp);
16537         ddi_dma_free_handle(dma_hdp);
16538         *dma_hdp = NULL;
16539 }
```

```
**********************************************************
   83539 Mon Dec 22 09:51:45 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
First attempt at pulling 4310 fix from Andy Giles tree
**********************************************************
______unchanged_portion_omitted_

1080 /*
1081  * NOTE: We should be able to queue TM requests in the controller to make this
1082  * a lot faster.  If resetting all targets, for example, we can load the hi
1083  * priority queue with its limit and the controller will reply as they are
1084  * completed.  This way, we don't have to poll for one reply at a time.
1085  * Think about enhancing this later.
1086  */
1087 int
1088 mptsas_ioc_task_management(mptsas_t *mpt, int task_type, uint16_t dev_handle,
1089         int lun, uint8_t *reply, uint32_t reply_size, int mode)
1090 {
1091         /*
1092          * In order to avoid allocating variables on the stack,
1093          * we make use of the pre-existing mptsas_cmd_t and
1094          * scsi_pkt which are included in the mptsas_t which
1095          * is passed to this routine.
1096          */

1098         pMpi2SCSITaskManagementRequest_t        task;
1099         int                                     rval = FALSE;
1100         mptsas_cmd_t                            *cmd;
1101         struct scsi_pkt                         *pkt;
1102         mptsas_slots_t                          *slots = mpt->m_active;
1103         uint64_t                                request_desc, i;
1104         pMPI2DefaultReply_t                     reply_msg;

1106         /*
1107          * Can't start another task management routine.
1108          */
1109         if (slots->m_slot[MPTSAS_TM_SLOT(mpt)] != NULL) {
1110                 mptsas_log(mpt, CE_WARN, "Can only start 1 task management"
1111                     " command at a time\n");
1112                 return (FALSE);
1113         }

1115         cmd = &(mpt->m_event_task_mgmt.m_event_cmd);
1116         pkt = &(mpt->m_event_task_mgmt.m_event_pkt);

1118         bzero((caddr_t)cmd, sizeof (*cmd));
1119         bzero((caddr_t)pkt, scsi_pkt_size());

1121         pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
1122         pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
1123         pkt->pkt_ha_private     = (opaque_t)cmd;
1124         pkt->pkt_flags          = (FLAG_NOINTR | FLAG_HEAD);
1125         pkt->pkt_time           = 60;
1126         pkt->pkt_address.a_target = dev_handle;
1127         pkt->pkt_address.a_lun = (uchar_t)lun;
1128         cmd->cmd_pkt            = pkt;
1129         cmd->cmd_scblen         = 1;
1130         cmd->cmd_flags          = CFLAG_TM_CMD;
1131         cmd->cmd_slot           = MPTSAS_TM_SLOT(mpt);

1133         slots->m_slot[MPTSAS_TM_SLOT(mpt)] = cmd;

1135         /*
1136          * Store the TM message in memory location corresponding to the TM slot
1137          * number.
1138          */
```



```
1139         task = (pMpi2SCSITaskManagementRequest_t)(mpt->m_req_frame +
1140             (mpt->m_req_frame_size * cmd->cmd_slot));
1141         bzero(task, mpt->m_req_frame_size);

1143         /*
1144          * form message for requested task
1145          */
1146         mptsas_init_std_hdr(mpt->m_acc_req_frame_hdl, task, dev_handle, lun, 0,
1147             MPI2_FUNCTION_SCSI_TASK_MGMT);

1149         /*
1150          * Set the task type
1151          */
1152         ddi_put8(mpt->m_acc_req_frame_hdl, &task->TaskType, task_type);

1154         /*
1155          * Send TM request using High Priority Queue.
1156          */
1157         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1158             DDI_DMA_SYNC_FORDEV);
1159         request_desc = (cmd->cmd_slot << 16) +
1160             MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY;
1161         MPTSAS_START_CMD(mpt, request_desc);
1162         rval = mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);

1164         if (pkt->pkt_reason == CMD_INCOMPLETE)
1165                 rval = FALSE;

1167         /*
1168          * If a reply frame was used and there is a reply buffer to copy the
1169          * reply data into, copy it.  If this fails, log a message, but don't
1170          * fail the TM request.
1171          */
1172         if (cmd->cmd_rfm && reply) {
1173                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
1174                     DDI_DMA_SYNC_FORCPU);
1175                 reply_msg = (pMPI2DefaultReply_t)
1176                     (mpt->m_reply_frame + (cmd->cmd_rfm -
1177                     (mpt->m_reply_frame_dma_addr & 0xffffffffu)));
1178                 if (reply_size > sizeof (MPI2_SCSI_TASK_MANAGE_REPLY)) {
1179                         reply_size = sizeof (MPI2_SCSI_TASK_MANAGE_REPLY);
1180                 }
1181                 mutex_exit(&mpt->m_mutex);
1182                 for (i = 0; i < reply_size; i++) {
1183                         if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
1184                             mode)) {
1185                                 mptsas_log(mpt, CE_WARN, "failed to copy out "
1186                                     "reply data for TM request");
1187                                 break;
1188                         }
1189                 }
1190                 mutex_enter(&mpt->m_mutex);
1191         }

1193         /*
1194          * clear the TM slot before returning
1195          */
1196         slots->m_slot[MPTSAS_TM_SLOT(mpt)] = NULL;

1198         /*
1199          * If we lost our task management command we need to reset the ioc
1200          * but we can't do it here as it's most likely we were called from
1201          * the taskq that mptsas_restart_ioc()->mptsas_flush_hba() needs to
1202          * wait for. So set a flag for mptsas_watch().
1199          * If we lost our task management command
1200          * we need to reset the ioc
```

```
1203          */
1204         if (rval == FALSE) {
1205                 mptsas_log(mpt, CE_WARN, "mptsas_ioc_task_management failed "
1206                     "schedule reset in watch!");
1207                 mpt->m_softstate |= MPTSAS_SS_RESET_INWATCH;
1204                     "try to reset ioc to recovery!");
1205                 mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1206                 if (mptsas_restart_ioc(mpt)) {
1207                         mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1208                         rval = FAILED;
1209                 }
1208         }
1209
1210         return (rval);
1211 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   45240 Mon Dec 22 09:51:45 2014
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
First attempt at pulling 4310 fix from Andy Giles tree
**********************************************************
_____unchanged_portion_omitted_
 940 #define MPTSAS_SIZE     (sizeof (struct mptsas) - \
 941                         sizeof (struct scsi_pkt) + scsi_pkt_size())
 942 /*
 943  * Only one of below two conditions is satisfied, we
 944  * think the target is associated to the iport and
 945  * allow call into mptsas_probe_lun().
 946  * 1. physicalsport == physport
 947  * 2. (phymask & (1 << physport)) == 0
 948  * The condition #2 is because LSI uses lowest PHY
 949  * number as the value of physical port when auto port
 950  * configuration.
 951  */
 952 #define IS_SAME_PORT(physicalport, physport, phymask, dynamicport) \
 953         ((physicalport == physport) || (dynamicport && (phymask & \
 954         (1 << physport))))

 956 _NOTE(MUTEX_PROTECTS_DATA(mptsas::m_mutex, mptsas))
 957 _NOTE(SCHEME_PROTECTS_DATA("safe sharing", mptsas::m_next))
 958 _NOTE(SCHEME_PROTECTS_DATA("stable data", mptsas::m_dip mptsas::m_tran))
 959 _NOTE(SCHEME_PROTECTS_DATA("stable data", mptsas::m_kmem_cache))
 960 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_io_dma_attr.dma_attr_sgllen))
 961 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_devid))
 962 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_productid))
 963 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_mpxio_enable))
 964 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_instance))

 966 /*
 967  * These should eventually migrate into the mpt header files
 968  * that may become the /kernel/misc/mpt module...
 969  */
 970 #define mptsas_init_std_hdr(hdl, mp, DevHandle, Lun, ChainOffset, Function) \
 971         mptsas_put_msg_DevHandle(hdl, mp, DevHandle); \
 972         mptsas_put_msg_ChainOffset(hdl, mp, ChainOffset); \
 973         mptsas_put_msg_Function(hdl, mp, Function); \
 974         mptsas_put_msg_Lun(hdl, mp, Lun)

 976 #define mptsas_put_msg_DevHandle(hdl, mp, val) \
 977         ddi_put16(hdl, &(mp)->DevHandle, (val))
 978 #define mptsas_put_msg_ChainOffset(hdl, mp, val) \
 979         ddi_put8(hdl, &(mp)->ChainOffset, (val))
 980 #define mptsas_put_msg_Function(hdl, mp, val) \
 981         ddi_put8(hdl, &(mp)->Function, (val))
 982 #define mptsas_put_msg_Lun(hdl, mp, val) \
 983         ddi_put8(hdl, &(mp)->LUN[1], (val))

 985 #define mptsas_get_msg_Function(hdl, mp) \
 986         ddi_get8(hdl, &(mp)->Function)

 988 #define mptsas_get_msg_MsgFlags(hdl, mp) \
 989         ddi_get8(hdl, &(mp)->MsgFlags)

 991 #define MPTSAS_ENABLE_DRWE(hdl) \
 992         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
 993                 MPI2_WRSEQ_FLUSH_KEY_VALUE); \
 994         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
 995                 MPI2_WRSEQ_1ST_KEY_VALUE); \
 996         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
 997                 MPI2_WRSEQ_2ND_KEY_VALUE); \
 998         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
 999                 MPI2_WRSEQ_3RD_KEY_VALUE); \
```



```
1000         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1001                 MPI2_WRSEQ_4TH_KEY_VALUE); \
1002         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1003                 MPI2_WRSEQ_5TH_KEY_VALUE); \
1004         ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1005                 MPI2_WRSEQ_6TH_KEY_VALUE);

1007 /*
1008  * m_options flags
1009  */
1010 #define MPTSAS_OPT_PM           0x01    /* Power Management */

1012 /*
1013  * m_softstate flags
1014  */
1015 #define MPTSAS_SS_DRAINING              0x02
1016 #define MPTSAS_SS_QUIESCED              0x04
1017 #define MPTSAS_SS_MSG_UNIT_RESET        0x08
1018 #define MPTSAS_DID_MSG_UNIT_RESET       0x10
1019 #define MPTSAS_SS_RESET_INWATCH         0x20
1020 #define MPTSAS_SS_MUR_INWATCH           0x40
1021 #endif /* ! codereview */

1023 /*
1024  * regspec defines.
1025  */
1026 #define CONFIG_SPACE    0       /* regset[0] - configuration space */
1027 #define IO_SPACE        1       /* regset[1] - used for i/o mapped device */
1028 #define MEM_SPACE       2       /* regset[2] - used for memory mapped device */
1029 #define BASE_REG2       3       /* regset[3] - used for 875 scripts ram */

1031 /*
1032  * Handy constants
1033  */
1034 #define FALSE           0
1035 #define TRUE            1
1036 #define UNDEFINED       -1
1037 #define FAILED          -2

1039 /*
1040  * power management.
1041  */
1042 #define MPTSAS_POWER_ON(mpt) { \
1043         pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset, \
1044             PCI_PMCSR_D0); \
1045         delay(drv_usectohz(10000)); \
1046         (void) pci_restore_config_regs(mpt->m_dip); \
1047         mptsas_setup_cmd_reg(mpt); \
1048 }

1050 #define MPTSAS_POWER_OFF(mpt) { \
1051         (void) pci_save_config_regs(mpt->m_dip); \
1052         pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset, \
1053             PCI_PMCSR_D3HOT); \
1054         mpt->m_power_level = PM_LEVEL_D3; \
1055 }

1057 /*
1058  * inq_dtype:
1059  * Bits 5 through 7 are the Peripheral Device Qualifier
1060  * 001b: device not connected to the LUN
1061  * Bits 0 through 4 are the Peripheral Device Type
1062  * 1fh: Unknown or no device type
1063  *
1064  * Although the inquiry may return success, the following value
1065  * means no valid LUN connected.
```

```
 */
#define MPTSAS_VALID_LUN(sd_inq) \
        (((sd_inq->inq_dtype & 0xe0) != 0x20) && \
        ((sd_inq->inq_dtype & 0x1f) != 0x1f))

/*
 * Default is to have 10 retries on receiving QFULL status and
 * each retry to be after 100 ms.
 */
#define QFULL_RETRIES           10
#define QFULL_RETRY_INTERVAL    100

/*
 * Handy macros
 */
#define Tgt(sp) ((sp)->cmd_pkt->pkt_address.a_target)
#define Lun(sp) ((sp)->cmd_pkt->pkt_address.a_lun)

#define IS_HEX_DIGIT(n) (((n) >= '0' && (n) <= '9') || \
        ((n) >= 'a' && (n) <= 'f') || ((n) >= 'A' && (n) <= 'F'))

/*
 * poll time for mptsas_pollret() and mptsas_wait_intr()
 */
#define MPTSAS_POLL_TIME        30000   /* 30 seconds */

/*
 * default time for mptsas_do_passthru
 */
#define MPTSAS_PASS_THRU_TIME_DEFAULT   60      /* 60 seconds */

/*
 * macro to return the effective address of a given per-target field
 */
#define EFF_ADDR(start, offset)         ((start) + (offset))

#define SDEV2ADDR(devp)         (&((devp)->sd_address))
#define SDEV2TRAN(devp)         ((devp)->sd_address.a_hba_tran)
#define PKT2TRAN(pkt)           ((pkt)->pkt_address.a_hba_tran)
#define ADDR2TRAN(ap)           ((ap)->a_hba_tran)
#define DIP2TRAN(dip)           (ddi_get_driver_private(dip))


#define TRAN2MPT(hba)           ((mptsas_t *)(hba)->tran_hba_private)
#define DIP2MPT(dip)            (TRAN2MPT((scsi_hba_tran_t *)DIP2TRAN(dip)))
#define SDEV2MPT(sd)            (TRAN2MPT(SDEV2TRAN(sd)))
#define PKT2MPT(pkt)            (TRAN2MPT(PKT2TRAN(pkt)))

#define ADDR2MPT(ap)            (TRAN2MPT(ADDR2TRAN(ap)))

#define POLL_TIMEOUT            (2 * SCSI_POLL_TIMEOUT * 1000000)
#define SHORT_POLL_TIMEOUT      (1000000)       /* in usec, about 1 secs */
#define MPTSAS_QUIESCE_TIMEOUT  1               /* 1 sec */
#define MPTSAS_PM_IDLE_TIMEOUT  60              /* 60 seconds */

#define MPTSAS_GET_ISTAT(mpt)  (ddi_get32((mpt)->m_datap, \
                        &(mpt)->m_reg->HostInterruptStatus))

#define MPTSAS_SET_SIGP(P) \
                ClrSetBits(mpt->m_devaddr + NREG_ISTAT, 0, NB_ISTAT_SIGP)

#define MPTSAS_RESET_SIGP(P) (void) ddi_get8(mpt->m_datap, \
                        (uint8_t *)(mpt->m_devaddr + NREG_CTEST2))

#define MPTSAS_GET_INTCODE(P) (ddi_get32(mpt->m_datap, \
                        (uint32_t *)(mpt->m_devaddr + NREG_DSPS)))
```



```


#define MPTSAS_START_CMD(mpt, req_desc) \
        ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostLow,  \
            req_desc & 0xffffffffu);                                    \
        ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostHigh, \
            (req_desc >> 32) & 0xffffffffu);

#define INTPENDING(mpt) \
        (MPTSAS_GET_ISTAT(mpt) & MPI2_HIS_REPLY_DESCRIPTOR_INTERRUPT)

/*
 * Mask all interrupts to disable
 */
#define MPTSAS_DISABLE_INTR(mpt)        \
        ddi_put32((mpt)->m_datap, &(mpt)->m_reg->HostInterruptMask, \
            (MPI2_HIM_RIM | MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

/*
 * Mask Doorbell and Reset interrupts to enable reply desc int.
 */
#define MPTSAS_ENABLE_INTR(mpt) \
        ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, \
        (MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

#define MPTSAS_GET_NEXT_REPLY(mpt, index)  \
        &((uint64_t *)(void *)mpt->m_post_queue)[index]

#define MPTSAS_GET_NEXT_FRAME(mpt, SMID) \
        (mpt->m_req_frame + (mpt->m_req_frame_size * SMID))

#define ClrSetBits32(hdl, reg, clr, set) \
        ddi_put32(hdl, (reg), \
            ((ddi_get32(mpt->m_datap, (reg)) & ~(clr)) | (set)))

#define ClrSetBits(reg, clr, set) \
        ddi_put8(mpt->m_datap, (uint8_t *)(reg), \
                ((ddi_get8(mpt->m_datap, (uint8_t *)(reg)) & ~(clr)) | (set)))

#define MPTSAS_WAITQ_RM(mpt, cmdp)      \
        if ((cmdp = mpt->m_waitq) != NULL) { \
                /* If the queue is now empty fix the tail pointer */    \
                if ((mpt->m_waitq = cmdp->cmd_linkp) == NULL) \
                        mpt->m_waitqtail = &mpt->m_waitq; \
                cmdp->cmd_linkp = NULL; \
                cmdp->cmd_queued = FALSE; \
        }

#define MPTSAS_TX_WAITQ_RM(mpt, cmdp)   \
        if ((cmdp = mpt->m_tx_waitq) != NULL) { \
                /* If the queue is now empty fix the tail pointer */    \
                if ((mpt->m_tx_waitq = cmdp->cmd_linkp) == NULL) \
                        mpt->m_tx_waitqtail = &mpt->m_tx_waitq; \
                cmdp->cmd_linkp = NULL; \
                cmdp->cmd_queued = FALSE; \
        }

/*
 * defaults for the global properties
 */
#define DEFAULT_SCSI_OPTIONS    SCSI_OPTIONS_DR
#define DEFAULT_TAG_AGE_LIMIT   2
#define DEFAULT_WD_TICK         1

/*
 * invalid hostid.
```

```
 */
#define MPTSAS_INVALID_HOSTID  -1

/*
 * Get/Set hostid from SCSI port configuration page
 */
#define MPTSAS_GET_HOST_ID(configuration) (configuration & 0xFF)
#define MPTSAS_SET_HOST_ID(hostid) (hostid | ((1 << hostid) << 16))

/*
 * Config space.
 */
#define MPTSAS_LATENCY_TIMER    0x40

/*
 * Offset to firmware version
 */
#define MPTSAS_FW_VERSION_OFFSET        9

/*
 * Offset and masks to get at the ProductId field
 */
#define MPTSAS_FW_PRODUCTID_OFFSET      8
#define MPTSAS_FW_PRODUCTID_MASK        0xFFFF0000
#define MPTSAS_FW_PRODUCTID_SHIFT       16

/*
 * Subsystem ID for HBAs.
 */
#define MPTSAS_HBA_SUBSYSTEM_ID    0x10C0
#define MPTSAS_RHEA_SUBSYSTEM_ID        0x10B0

/*
 * reset delay tick
 */
#define MPTSAS_WATCH_RESET_DELAY_TICK 50        /* specified in milli seconds */

/*
 * Ioc reset return values
 */
#define MPTSAS_RESET_FAIL       -1
#define MPTSAS_NO_RESET         0
#define MPTSAS_SUCCESS_HARDRESET        1
#define MPTSAS_SUCCESS_MUR      2

/*
 * throttle support.
 */
#define MAX_THROTTLE    32
#define HOLD_THROTTLE   0
#define DRAIN_THROTTLE  -1
#define QFULL_THROTTLE  -2

/*
 * Passthrough/config request flags
 */
#define MPTSAS_DATA_ALLOCATED           0x0001
#define MPTSAS_DATAOUT_ALLOCATED        0x0002
#define MPTSAS_REQUEST_POOL_CMD         0x0004
#define MPTSAS_ADDRESS_REPLY            0x0008
#define MPTSAS_CMD_TIMEOUT              0x0010

/*
 * response code tlr flag
 */
#define MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF       0x02
```



```
/*
 * System Events
 */
#ifndef DDI_VENDOR_LSI
#define DDI_VENDOR_LSI  "LSI"
#endif  /* DDI_VENDOR_LSI */

/*
 * Shared functions
 */
int mptsas_save_cmd(struct mptsas *mpt, struct mptsas_cmd *cmd);
void mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_log(struct mptsas *mpt, int level, char *fmt, ...);
int mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime);
int mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)());
int mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
        uint8_t type, int mode);
int mptsas_check_flash(mptsas_t *mpt, caddr_t origfile, uint32_t size,
        uint8_t type, int mode);
int mptsas_download_firmware();
int mptsas_can_download_firmware();
int mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_statep);
void mptsas_dma_free(mptsas_dma_alloc_state_t *dma_statep);
mptsas_phymask_t mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport);
void mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd);
int mptsas_check_acc_handle(ddi_acc_handle_t handle);
int mptsas_check_dma_handle(ddi_dma_handle_t handle);
void mptsas_fm_ereport(mptsas_t *mpt, char *detail);
int mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
    ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
    uint32_t alloc_size, ddi_dma_cookie_t *cookiep);
void mptsas_dma_addr_destroy(ddi_dma_handle_t *, ddi_acc_handle_t *);

/*
 * impl functions
 */
int mptsas_ioc_wait_for_response(mptsas_t *mpt);
int mptsas_ioc_wait_for_doorbell(mptsas_t *mpt);
int mptsas_ioc_reset(mptsas_t *mpt, int);
int mptsas_send_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
    ddi_acc_handle_t accessp);
int mptsas_get_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
    ddi_acc_handle_t accessp);
int mptsas_send_config_request_msg(mptsas_t *mpt, uint8_t action,
    uint8_t pagetype, uint32_t pageaddress, uint8_t pagenumber,
    uint8_t pageversion, uint8_t pagelength, uint32_t SGEflagslength,
    uint64_t SGEaddress);
int mptsas_send_extended_config_request_msg(mptsas_t *mpt, uint8_t action,
    uint8_t extpagetype, uint32_t pageaddress, uint8_t pagenumber,
    uint8_t pageversion, uint16_t extpagelength,
    uint32_t SGEflagslength, uint64_t SGEaddress);

int mptsas_request_from_pool(mptsas_t *mpt, mptsas_cmd_t **cmd,
    struct scsi_pkt **pkt);
void mptsas_return_to_pool(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_destroy_ioc_event_cmd(mptsas_t *mpt);
void mptsas_start_config_page_access(mptsas_t *mpt, mptsas_cmd_t *cmd);
int mptsas_access_config_page(mptsas_t *mpt, uint8_t action, uint8_t page_type,
    uint8_t page_number, uint32_t page_address, int (*callback) (mptsas_t *,
    caddr_t, ddi_acc_handle_t, uint16_t, uint32_t, va_list), ...);

int mptsas_ioc_task_management(mptsas_t *mpt, int task_type,
    uint16_t dev_handle, int lun, uint8_t *reply, uint32_t reply_size,
    int mode);
```

```
1330 int mptsas_send_event_ack(mptsas_t *mpt, uint32_t event, uint32_t eventcntx);
1331 void mptsas_send_pending_event_ack(mptsas_t *mpt);
1332 void mptsas_set_throttle(struct mptsas *mpt, mptsas_target_t *ptgt, int what);
1333 int mptsas_restart_ioc(mptsas_t *mpt);
1334 void mptsas_update_driver_data(struct mptsas *mpt);
1335 uint64_t mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun);

1337 /*
1338  * init functions
1339  */
1340 int mptsas_ioc_get_facts(mptsas_t *mpt);
1341 int mptsas_ioc_get_port_facts(mptsas_t *mpt, int port);
1342 int mptsas_ioc_enable_port(mptsas_t *mpt);
1343 int mptsas_ioc_enable_event_notification(mptsas_t *mpt);
1344 int mptsas_ioc_init(mptsas_t *mpt);

1346 /*
1347  * configuration pages operation
1348  */
1349 int mptsas_get_sas_device_page0(mptsas_t *mpt, uint32_t page_address,
1350     uint16_t *dev_handle, uint64_t *sas_wwn, uint32_t *dev_info,
1351     uint8_t *physport, uint8_t *phynum, uint16_t *pdevhandle,
1352     uint16_t *slot_num, uint16_t *enclosure, uint16_t *io_flags);
1353 int mptsas_get_sas_io_unit_page(mptsas_t *mpt);
1354 int mptsas_get_sas_io_unit_page_hndshk(mptsas_t *mpt);
1355 int mptsas_get_sas_expander_page0(mptsas_t *mpt, uint32_t page_address,
1356     mptsas_smp_t *info);
1357 int mptsas_set_ioc_params(mptsas_t *mpt);
1358 int mptsas_get_manufacture_page5(mptsas_t *mpt);
1359 int mptsas_get_sas_port_page0(mptsas_t *mpt, uint32_t page_address,
1360     uint64_t *sas_wwn, uint8_t *portwidth);
1361 int mptsas_get_bios_page3(mptsas_t *mpt,  uint32_t *bios_version);
1362 int
1363 mptsas_get_sas_phy_page0(mptsas_t *mpt, uint32_t page_address,
1364     smhba_info_t *info);
1365 int
1366 mptsas_get_sas_phy_page1(mptsas_t *mpt, uint32_t page_address,
1367     smhba_info_t *info);
1368 int
1369 mptsas_get_manufacture_page0(mptsas_t *mpt);
1370 void
1371 mptsas_create_phy_stats(mptsas_t *mpt, char *iport, dev_info_t *dip);
1372 void mptsas_destroy_phy_stats(mptsas_t *mpt);
1373 int mptsas_smhba_phy_init(mptsas_t *mpt);
1374 /*
1375  * RAID functions
1376  */
1377 int mptsas_get_raid_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol);
1378 int mptsas_get_raid_info(mptsas_t *mpt);
1379 int mptsas_get_physdisk_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol,
1380     uint8_t physdisknum);
1381 int mptsas_delete_volume(mptsas_t *mpt, uint16_t volid);
1382 void mptsas_raid_action_system_shutdown(mptsas_t *mpt);

1384 #define MPTSAS_IOCSTATUS(status) (status & MPI2_IOCSTATUS_MASK)
1385 /*
1386  * debugging.
1387  * MPTSAS_DBGLOG_LINECNT must be a power of 2.
1388  */
1389 #define MPTSAS_DBGLOG_LINECNT   128
1390 #define MPTSAS_DBGLOG_LINELEN   256
1391 #define MPTSAS_DBGLOG_BUFSIZE   (MPTSAS_DBGLOG_LINECNT * MPTSAS_DBGLOG_LINELEN)

1393 #if defined(MPTSAS_DEBUG)

1395 extern uint32_t mptsas_debugprt_flags;
```



```
1396 extern uint32_t mptsas_debuglog_flags;

1398 void mptsas_printf(char *fmt, ...);
1399 void mptsas_debug_log(char *fmt, ...);

1401 #define MPTSAS_DBGPR(m, args)   \
1402         if (mptsas_debugprt_flags & (m)) \
1403                 mptsas_printf args;   \
1404         if (mptsas_debuglog_flags & (m)) \
1405                 mptsas_debug_log args
1406 #else   /* ! defined(MPTSAS_DEBUG) */
1407 #define MPTSAS_DBGPR(m, args)
1408 #endif  /* defined(MPTSAS_DEBUG) */

1410 #define NDBG0(args)     MPTSAS_DBGPR(0x01, args)        /* init */
1411 #define NDBG1(args)     MPTSAS_DBGPR(0x02, args)        /* normal running */
1412 #define NDBG2(args)     MPTSAS_DBGPR(0x04, args)        /* property handling */
1413 #define NDBG3(args)     MPTSAS_DBGPR(0x08, args)        /* pkt handling */

1415 #define NDBG4(args)     MPTSAS_DBGPR(0x10, args)        /* kmem alloc/free */
1416 #define NDBG5(args)     MPTSAS_DBGPR(0x20, args)        /* polled cmds */
1417 #define NDBG6(args)     MPTSAS_DBGPR(0x40, args)        /* interrupts */
1418 #define NDBG7(args)     MPTSAS_DBGPR(0x80, args)        /* queue handling */

1420 #define NDBG8(args)     MPTSAS_DBGPR(0x0100, args)      /* arq */
1421 #define NDBG9(args)     MPTSAS_DBGPR(0x0200, args)      /* Tagged Q'ing */
1422 #define NDBG10(args)    MPTSAS_DBGPR(0x0400, args)      /* halting chip */
1423 #define NDBG11(args)    MPTSAS_DBGPR(0x0800, args)      /* power management */

1425 #define NDBG12(args)    MPTSAS_DBGPR(0x1000, args)      /* enumeration */
1426 #define NDBG13(args)    MPTSAS_DBGPR(0x2000, args)      /* configuration page */
1427 #define NDBG14(args)    MPTSAS_DBGPR(0x4000, args)      /* LED control */
1428 #define NDBG15(args)    MPTSAS_DBGPR(0x8000, args)      /* Passthrough */

1430 #define NDBG16(args)    MPTSAS_DBGPR(0x010000, args)    /* SAS Broadcasts */
1431 #define NDBG17(args)    MPTSAS_DBGPR(0x020000, args)    /* scatter/gather */
1432 #define NDBG18(args)    MPTSAS_DBGPR(0x040000, args)
1433 #define NDBG19(args)    MPTSAS_DBGPR(0x080000, args)    /* handshaking */

1435 #define NDBG20(args)    MPTSAS_DBGPR(0x100000, args)    /* events */
1436 #define NDBG21(args)    MPTSAS_DBGPR(0x200000, args)    /* dma */
1437 #define NDBG22(args)    MPTSAS_DBGPR(0x400000, args)    /* reset */
1438 #define NDBG23(args)    MPTSAS_DBGPR(0x800000, args)    /* abort */

1440 #define NDBG24(args)    MPTSAS_DBGPR(0x1000000, args)   /* capabilities */
1441 #define NDBG25(args)    MPTSAS_DBGPR(0x2000000, args)   /* flushing */
1442 #define NDBG26(args)    MPTSAS_DBGPR(0x4000000, args)
1443 #define NDBG27(args)    MPTSAS_DBGPR(0x8000000, args)   /* passthrough */

1445 #define NDBG28(args)    MPTSAS_DBGPR(0x10000000, args)  /* hotplug */
1446 #define NDBG29(args)    MPTSAS_DBGPR(0x20000000, args)  /* timeouts */
1447 #define NDBG30(args)    MPTSAS_DBGPR(0x40000000, args)  /* mptsas_watch */
1448 #define NDBG31(args)    MPTSAS_DBGPR(0x80000000, args)  /* negotations */

1450 /*
1451  * auto request sense
1452  */
1453 #define RQ_MAKECOM_COMMON(pkt, flag, cmd) \
1454         (pkt)->pkt_flags = (flag), \
1455         ((union scsi_cdb *)(pkt)->pkt_cdbp)->scc_cmd = (cmd), \
1456         ((union scsi_cdb *)(pkt)->pkt_cdbp)->scc_lun = \
1457             (pkt)->pkt_address.a_lun

1459 #define RQ_MAKECOM_G0(pkt, flag, cmd, addr, cnt) \
1460         RQ_MAKECOM_COMMON((pkt), (flag), (cmd)), \
1461         FORMG0ADDR(((union scsi_cdb *)(pkt)->pkt_cdbp), (addr)), \
```

```
1462         FORMG0COUNT(((union scsi_cdb *)(pkt)->pkt_cdbp), (cnt))


1465 #ifdef  __cplusplus
1466 }
1467 #endif

1469 #endif  /* _SYS_SCSI_ADAPTERS_MPTVAR_H */
```