

```

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

1024 /*
1025  * Notes:
1026  *   Set up all device state and allocate data structures,
1027  *   mutexes, condition variables, etc. for device operation.
1028  *   Add interrupts needed.
1029  *   Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
1030  */
1031 static int
1032 mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
1033 {
1034     mptsas_t          *mpt = NULL;
1035     int                instance, i, j;
1036     int                doneq_thread_num;
1037     char               intr_added = 0;
1038     char               map_setup = 0;
1039     char               config_setup = 0;
1040     char               hba_attach_setup = 0;
1041     char               smp_attach_setup = 0;
1042     char               mutex_init_done = 0;
1043     char               event_taskq_create = 0;
1044     char               dr_taskq_create = 0;
1045     char               doneq_thread_create = 0;
1046     char               added_watchdog = 0;
1047     scsi_hba_tran_t   *hba_tran;
1048     uint_t             mem_bar = MEM_SPACE;
1049     int                rval = DDI_FAILURE;

1051     /* CONSTCOND */
1052     ASSERT(NO_COMPETING_THREADS);

1054     if (scsi_hba_iport_unit_address(dip)) {
1055         return (mptsas_iport_attach(dip, cmd));
1056     }

1058     switch (cmd) {
1059     case DDI_ATTACH:
1060         break;

1062     case DDI_RESUME:
1063         if ((hba_tran = ddi_get_driver_private(dip)) == NULL)
1064             return (DDI_FAILURE);

1066         mpt = TRAN2MPT(hba_tran);

1068         if (!mpt) {
1069             return (DDI_FAILURE);
1070         }

1072         /*
1073          * Reset hardware and softc to "no outstanding commands"
1074          * Note that a check condition can result on first command
1075          * to a target.
1076          */
1077         mutex_enter(&mpt->m_mutex);

1079         /*
1080          * raise power.
1081          */
1082         if (mpt->m_options & MPTSAS_OPT_PM) {

```

```

1083         mutex_exit(&mpt->m_mutex);
1084         (void) pm_busy_component(dip, 0);
1085         rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
1086         if (rval == DDI_SUCCESS) {
1087             mutex_enter(&mpt->m_mutex);
1088         } else {
1089             /*
1090              * The pm_raise_power() call above failed,
1091              * and that can only occur if we were unable
1092              * to reset the hardware. This is probably
1093              * due to unhealthy hardware, and because
1094              * important filesystems (such as the root
1095              * filesystem) could be on the attached disks,
1096              * it would not be a good idea to continue,
1097              * as we won't be entirely certain we are
1098              * writing correct data. So we panic() here
1099              * to not only prevent possible data corruption,
1100              * but to give developers or end users a hope
1101              * of identifying and correcting any problems.
1102              */
1103             fm_panic("mptsas could not reset hardware "
1104                    "during resume");
1105         }
1106     }

1108     mpt->m_suspended = 0;

1110     /*
1111      * Reinitialize ioc
1112      */
1113     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1114     if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
1115         mutex_exit(&mpt->m_mutex);
1116         if (mpt->m_options & MPTSAS_OPT_PM) {
1117             (void) pm_idle_component(dip, 0);
1118         }
1119         fm_panic("mptsas init chip fail during resume");
1120     }
1121     /*
1122      * mptsas_update_driver_data needs interrupts so enable them
1123      * first.
1124      */
1125     MPTSAS_ENABLE_INTR(mpt);
1126     mptsas_update_driver_data(mpt);

1128     /* start requests, if possible */
1129     mptsas_restart_hba(mpt);

1131     mutex_exit(&mpt->m_mutex);

1133     /*
1134      * Restart watch thread
1135      */
1136     mutex_enter(&mptsas_global_mutex);
1137     if (mptsas_timeout_id == 0) {
1138         mptsas_timeout_id = timeout(mptsas_watch, NULL,
1139                                     mptsas_tick);
1140         mptsas_timeouts_enabled = 1;
1141     }
1142     mutex_exit(&mptsas_global_mutex);

1144     /* report idle status to pm framework */
1145     if (mpt->m_options & MPTSAS_OPT_PM) {
1146         (void) pm_idle_component(dip, 0);
1147     }

```

```

1149         return (DDI_SUCCESS);
1151     default:
1152         return (DDI_FAILURE);
1154 }
1156 instance = ddi_get_instance(dip);
1158 /*
1159  * Allocate softc information.
1160  */
1161 if (ddi_soft_state_zalloc(mptsas_state, instance) != DDI_SUCCESS) {
1162     mptsas_log(NULL, CE_WARN,
1163         "mptsas%d: cannot allocate soft state", instance);
1164     goto fail;
1165 }
1167 mpt = ddi_get_soft_state(mptsas_state, instance);
1169 if (mpt == NULL) {
1170     mptsas_log(NULL, CE_WARN,
1171         "mptsas%d: cannot get soft state", instance);
1172     goto fail;
1173 }
1175 /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
1176 scsi_size_clean(dip);
1178 mpt->m_dip = dip;
1179 mpt->m_instance = instance;
1181 /* Make a per-instance copy of the structures */
1182 mpt->m_io_dma_attr = mptsas_dma_attrs64;
1183 mpt->m_msg_dma_attr = mptsas_dma_attrs;
1184 mpt->m_reg_acc_attr = mptsas_dev_attr;
1185 mpt->m_dev_acc_attr = mptsas_dev_attr;
1187 /*
1188  * Initialize FMA
1189  */
1190 mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
1191     DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
1192     DDI_FM_EREPOR_T_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
1193     DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);
1195 mptsas_fm_init(mpt);
1197 if (mptsas_alloc_handshake_msg(mpt,
1198     sizeof (Mpi2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
1199     mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
1200     goto fail;
1201 }
1203 /*
1204  * Setup configuration space
1205  */
1206 if (mptsas_config_space_init(mpt) == FALSE) {
1207     mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
1208     goto fail;
1209 }
1210 config_setup++;
1212 if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
1213     0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
1214     mptsas_log(mpt, CE_WARN, "map setup failed");

```

```

1215         goto fail;
1216     }
1217     map_setup++;
1219     /*
1220     * A taskq is created for dealing with the event handler
1221     */
1222     if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
1223         1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1224         mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
1225         goto fail;
1226     }
1227     event_taskq_create++;
1229     /*
1230     * A taskq is created for dealing with dr events
1231     */
1232     if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
1233         "mptsas_dr_taskq",
1234         1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1235         mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
1236             "failed");
1237         goto fail;
1238     }
1239     dr_taskq_create++;
1241     mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1242         0, "mptsas_doneq_thread_threshold_prop", 10);
1243     mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1244         0, "mptsas_doneq_length_threshold_prop", 8);
1245     mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1246         0, "mptsas_doneq_thread_n_prop", 8);
1248     if (mpt->m_doneq_thread_n) {
1249         cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
1250         mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);
1252         mutex_enter(&mpt->m_doneq_mutex);
1253         mpt->m_doneq_thread_id =
1254             kmem_zalloc(sizeof (mptsas_doneq_thread_list_t)
1255                 * mpt->m_doneq_thread_n, KM_SLEEP);
1257         for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1258             cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
1259                 CV_DRIVER, NULL);
1260             mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
1261                 MUTEX_DRIVER, NULL);
1262             mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1263             mpt->m_doneq_thread_id[j].flag |=
1264                 MPTSAS_DONEQ_THREAD_ACTIVE;
1265             mpt->m_doneq_thread_id[j].arg.mpt = mpt;
1266             mpt->m_doneq_thread_id[j].arg.t = j;
1267             mpt->m_doneq_thread_id[j].threadp =
1268                 thread_create(NULL, 0, mptsas_doneq_thread,
1269                     &mpt->m_doneq_thread_id[j].arg,
1270                     0, &p0, TS_RUN, minclsyspri);
1271             mpt->m_doneq_thread_id[j].donetail =
1272                 &mpt->m_doneq_thread_id[j].doneq;
1273             mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1274         }
1275         mutex_exit(&mpt->m_doneq_mutex);
1276         doneq_thread_create++;
1277     }
1279     /*
1280     * Disable hardware interrupt since we're not ready to

```

```

1281     * handle it yet.
1282     */
1283     MPTSAS_DISABLE_INTR(mpt);
1284     if (mptsas_register_intrs(mpt) == FALSE)
1285         goto fail;
1286     intr_added++;

1288     /* Initialize mutex used in interrupt handler */
1289     mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
1290              DDI_INTR_PRI(mpt->m_intr_pri));
1291     mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
1292     mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
1293              DDI_INTR_PRI(mpt->m_intr_pri));
1294     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1295         mutex_init(&mpt->m_phy_info[i].smhba_info.phy_mutex,
1296                  NULL, MUTEX_DRIVER,
1297                  DDI_INTR_PRI(mpt->m_intr_pri));
1298     }

1300     cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
1301     cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
1302     cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
1303     cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
1304     cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
1305     mutex_init_done++;

1307     mutex_enter(&mpt->m_mutex);
1308     /*
1309     * Initialize power management component
1310     */
1311     if (mpt->m_options & MPTSAS_OPT_PM) {
1312         if (mptsas_init_pm(mpt)) {
1313             mutex_exit(&mpt->m_mutex);
1314             mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
1315                      "failed");
1316             goto fail;
1317         }
1318     }

1320     /*
1321     * Initialize chip using Message Unit Reset, if allowed
1322     */
1323     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1324     if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
1325         mutex_exit(&mpt->m_mutex);
1326         mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
1327         goto fail;
1328     }

1330     mpt->m_targets = reflash_create(MPTSAS_TARGET_BUCKET_COUNT,
1331                                  mptsas_target_addr_hash, mptsas_target_addr_cmp,
1332                                  mptsas_target_free, sizeof(mptsas_target_t),
1333                                  offsetof(mptsas_target_t, m_link),
1334                                  offsetof(mptsas_target_t, m_addr), KM_SLEEP);

1336 #endif /* ! codereview */
1337     /*
1338     * Fill in the phy_info structure and get the base WWID
1339     */
1340     if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
1341         mptsas_log(mpt, CE_WARN,
1342                  "mptsas_get_manufacture_page5 failed!");
1343         goto fail;
1344     }

1346     if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {

```

```

1347         mptsas_log(mpt, CE_WARN,
1348                  "mptsas_get_sas_io_unit_page_hndshk failed!");
1349         goto fail;
1350     }

1352     if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
1353         mptsas_log(mpt, CE_WARN,
1354                  "mptsas_get_manufacture_page0 failed!");
1355         goto fail;
1356     }

1358     mutex_exit(&mpt->m_mutex);

1360     /*
1361     * Register the iport for multiple port HBA
1362     */
1363     mptsas_iport_register(mpt);

1365     /*
1366     * initialize SCSI HBA transport structure
1367     */
1368     if (mptsas_hba_setup(mpt) == FALSE)
1369         goto fail;
1370     hba_attach_setup++;

1372     if (mptsas_smp_setup(mpt) == FALSE)
1373         goto fail;
1374     smp_attach_setup++;

1376     if (mptsas_cache_create(mpt) == FALSE)
1377         goto fail;

1379     mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
1380                                              dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
1381     if (mpt->m_scsi_reset_delay == 0) {
1382         mptsas_log(mpt, CE_NOTE,
1383                  "scsi_reset_delay of 0 is not recommended,"
1384                  " resetting to SCSI_DEFAULT_RESET_DELAY\n");
1385         mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
1386     }

1388     /*
1389     * Initialize the wait and done FIFO queue
1390     */
1391     mpt->m_donetail = &mpt->m_doneq;
1392     mpt->m_waitqtail = &mpt->m_waitq;
1393     mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
1394     mpt->m_tx_draining = 0;

1396     /*
1397     * ioc cmd queue initialize
1398     */
1399     mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
1400     mpt->m_dev_handle = 0xFFFF;

1402     MPTSAS_ENABLE_INTR(mpt);

1404     /*
1405     * enable event notification
1406     */
1407     mutex_enter(&mpt->m_mutex);
1408     if (mptsas_ioc_enable_event_notification(mpt)) {
1409         mutex_exit(&mpt->m_mutex);
1410         goto fail;
1411     }
1412     mutex_exit(&mpt->m_mutex);

```

```

1414  /*
1415  * used for mptsas_watch
1416  */
1417  mptsas_list_add(mpt);

1419  mutex_enter(&mptsas_global_mutex);
1420  if (mptsas_timeouts_enabled == 0) {
1421      mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
1422          dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);

1424      mptsas_tick = mptsas_scsi_watchdog_tick *
1425          drv_usecstohz((clock_t)1000000);

1427      mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
1428      mptsas_timeouts_enabled = 1;
1429  }
1430  mutex_exit(&mptsas_global_mutex);
1431  added_watchdog++;

1433  /*
1434  * Initialize PHY info for smhba.
1435  * This requires watchdog to be enabled otherwise if interrupts
1436  * don't work the system will hang.
1437  */
1438  if (mptsas_smhba_setup(mpt)) {
1439      mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
1440          "failed");
1441      goto fail;
1442  }

1444  /* Check all dma handles allocated in attach */
1445  if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
1446      != DDI_SUCCESS) ||
1447      (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
1448      != DDI_SUCCESS) ||
1449      (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
1450      != DDI_SUCCESS) ||
1451      (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
1452      != DDI_SUCCESS) ||
1453      (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
1454      != DDI_SUCCESS)) {
1455      goto fail;
1456  }

1458  /* Check all acc handles allocated in attach */
1459  if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
1460      (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
1461      != DDI_SUCCESS) ||
1462      (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
1463      != DDI_SUCCESS) ||
1464      (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
1465      != DDI_SUCCESS) ||
1466      (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
1467      != DDI_SUCCESS) ||
1468      (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
1469      != DDI_SUCCESS) ||
1470      (mptsas_check_acc_handle(mpt->m_config_handle)
1471      != DDI_SUCCESS)) {
1472      goto fail;
1473  }

1475  /*
1476  * After this point, we are not going to fail the attach.
1477  */

```

```

1479  /* Print message of HBA present */
1480  ddi_report_dev(dip);

1482  /* report idle status to pm framework */
1483  if (mpt->m_options & MPTSAS_OPT_PM) {
1484      (void) pm_idle_component(dip, 0);
1485  }

1487  return (DDI_SUCCESS);

1489 fail:
1490  mptsas_log(mpt, CE_WARN, "attach failed");
1491  mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
1492  ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
1493  if (mpt) {
1494      /* deallocate in reverse order */
1495      if (added_watchdog) {
1496          mptsas_list_del(mpt);
1497          mutex_enter(&mptsas_global_mutex);

1499          if (mptsas_timeout_id && (mptsas_head == NULL)) {
1500              timeout_id_t tid = mptsas_timeout_id;
1501              mptsas_timeouts_enabled = 0;
1502              mptsas_timeout_id = 0;
1503              mutex_exit(&mptsas_global_mutex);
1504              (void) untimeout(tid);
1505              mutex_enter(&mptsas_global_mutex);
1506          }
1507          mutex_exit(&mptsas_global_mutex);
1508      }

1510      mptsas_cache_destroy(mpt);

1512      if (smp_attach_setup) {
1513          mptsas_smp_teardown(mpt);
1514      }
1515      if (hba_attach_setup) {
1516          mptsas_hba_teardown(mpt);
1517      }

1519      if (mpt->m_targets)
1520          rehash_destroy(mpt->m_targets);
1521      if (mpt->m_smp_targets)
1522          rehash_destroy(mpt->m_smp_targets);

1524      if (mpt->m_active) {
1525          mptsas_free_active_slots(mpt);
1526      }
1527      if (intr_added) {
1528          mptsas_unregister_intrs(mpt);
1529      }

1531      if (doneq_thread_create) {
1532          mutex_enter(&mpt->m_doneq_mutex);
1533          doneq_thread_num = mpt->m_doneq_thread_n;
1534          for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1535              mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1536              mpt->m_doneq_thread_id[j].flag &=
1537                  (~MPTSAS_DONEQ_THREAD_ACTIVE);
1538              cv_signal(&mpt->m_doneq_thread_id[j].cv);
1539              mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1540          }
1541          while (mpt->m_doneq_thread_n) {
1542              cv_wait(&mpt->m_doneq_thread_cv,
1543                  &mpt->m_doneq_mutex);
1544          }

```

```

1545         for (j = 0; j < doneq_thread_num; j++) {
1546             cv_destroy(&mpt->m_doneq_thread_id[j].cv);
1547             mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
1548         }
1549         kmem_free(mpt->m_doneq_thread_id,
1550             sizeof (mptsas_doneq_thread_list_t)
1551             * doneq_thread_num);
1552         mutex_exit(&mpt->m_doneq_mutex);
1553         cv_destroy(&mpt->m_doneq_thread_cv);
1554         mutex_destroy(&mpt->m_doneq_mutex);
1555     }
1556     if (event_taskq_create) {
1557         ddi_taskq_destroy(mpt->m_event_taskq);
1558     }
1559     if (dr_taskq_create) {
1560         ddi_taskq_destroy(mpt->m_dr_taskq);
1561     }
1562     if (mutex_init_done) {
1563         mutex_destroy(&mpt->m_tx_waitq_mutex);
1564         mutex_destroy(&mpt->m_passthru_mutex);
1565         mutex_destroy(&mpt->m_mutex);
1566         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1567             mutex_destroy(
1568                 &mpt->m_phy_info[i].smhba_info.phy_mutex);
1569         }
1570         cv_destroy(&mpt->m_cv);
1571         cv_destroy(&mpt->m_passthru_cv);
1572         cv_destroy(&mpt->m_fw_cv);
1573         cv_destroy(&mpt->m_config_cv);
1574         cv_destroy(&mpt->m_fw_diag_cv);
1575     }
1576
1577     if (map_setup) {
1578         mptsas_cfg_fini(mpt);
1579     }
1580     if (config_setup) {
1581         mptsas_config_space_fini(mpt);
1582     }
1583     mptsas_free_handshake_msg(mpt);
1584     mptsas_hba_fini(mpt);
1585
1586     mptsas_fm_fini(mpt);
1587     ddi_soft_state_free(mptsas_state, instance);
1588     ddi_prop_remove_all(dip);
1589 }
1590 return (DDI_FAILURE);
1591 }
1592
1593 static int
1594 mptsas_suspend(dev_info_t *devi)
1595 {
1596     mptsas_t      *mpt, *g;
1597     scsi_hba_tran_t *tran;
1598
1599     if (scsi_hba_iport_unit_address(devi)) {
1600         return (DDI_SUCCESS);
1601     }
1602
1603     if ((tran = ddi_get_driver_private(devi)) == NULL)
1604         return (DDI_SUCCESS);
1605
1606     mpt = TRAN2MPT(tran);
1607     if (!mpt) {
1608         return (DDI_SUCCESS);
1609     }

```

```

1611         mutex_enter(&mpt->m_mutex);
1612
1613         if (mpt->m_suspended++) {
1614             mutex_exit(&mpt->m_mutex);
1615             return (DDI_SUCCESS);
1616         }
1617
1618         /*
1619          * Cancel timeout threads for this mpt
1620          */
1621         if (mpt->m_quiesce_timeid) {
1622             timeout_id_t tid = mpt->m_quiesce_timeid;
1623             mpt->m_quiesce_timeid = 0;
1624             mutex_exit(&mpt->m_mutex);
1625             (void) untimeout(tid);
1626             mutex_enter(&mpt->m_mutex);
1627         }
1628
1629         if (mpt->m_restart_cmd_timeid) {
1630             timeout_id_t tid = mpt->m_restart_cmd_timeid;
1631             mpt->m_restart_cmd_timeid = 0;
1632             mutex_exit(&mpt->m_mutex);
1633             (void) untimeout(tid);
1634             mutex_enter(&mpt->m_mutex);
1635         }
1636
1637         mutex_exit(&mpt->m_mutex);
1638
1639         (void) pm_idle_component(mpt->m_dip, 0);
1640
1641         /*
1642          * Cancel watch threads if all mpts suspended
1643          */
1644         rw_enter(&mptsas_global_rwlock, RW_WRITER);
1645         for (g = mptsas_head; g != NULL; g = g->m_next) {
1646             if (!g->m_suspended)
1647                 break;
1648         }
1649         rw_exit(&mptsas_global_rwlock);
1650
1651         mutex_enter(&mptsas_global_mutex);
1652         if (g == NULL) {
1653             timeout_id_t tid;
1654
1655             mptsas_timeouts_enabled = 0;
1656             if (mptsas_timeout_id) {
1657                 tid = mptsas_timeout_id;
1658                 mptsas_timeout_id = 0;
1659                 mutex_exit(&mptsas_global_mutex);
1660                 (void) untimeout(tid);
1661                 mutex_enter(&mptsas_global_mutex);
1662             }
1663             if (mptsas_reset_watch) {
1664                 tid = mptsas_reset_watch;
1665                 mptsas_reset_watch = 0;
1666                 mutex_exit(&mptsas_global_mutex);
1667                 (void) untimeout(tid);
1668                 mutex_enter(&mptsas_global_mutex);
1669             }
1670         }
1671         mutex_exit(&mptsas_global_mutex);
1672
1673         mutex_enter(&mpt->m_mutex);
1674
1675         /*
1676          * If this mpt is not in full power(PM_LEVEL_D0), just return.

```

```

1677     */
1678     if ((mpt->m_options & MPTSAS_OPT_PM) &&
1679         (mpt->m_power_level != PM_LEVEL_D0)) {
1680         mutex_exit(&mpt->m_mutex);
1681         return (DDI_SUCCESS);
1682     }
1684     /* Disable HBA interrupts in hardware */
1685     MPTSAS_DISABLE_INTR(mpt);
1686     /*
1687      * Send RAID action system shutdown to sync IR
1688      */
1689     mptsas_raid_action_system_shutdown(mpt);
1691     mutex_exit(&mpt->m_mutex);
1693     /* drain the taskq */
1694     ddi_taskq_wait(mpt->m_event_taskq);
1695     ddi_taskq_wait(mpt->m_dr_taskq);
1697     return (DDI_SUCCESS);
1698 }
1700 #ifdef __sparc
1701 /*ARGSUSED*/
1702 static int
1703 mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd)
1704 {
1705     mptsas_t      *mpt;
1706     scsi_hba_tran_t *tran;
1708     /*
1709      * If this call is for iport, just return.
1710      */
1711     if (scsi_hba_iport_unit_address(devi))
1712         return (DDI_SUCCESS);
1714     if ((tran = ddi_get_driver_private(devi)) == NULL)
1715         return (DDI_SUCCESS);
1717     if ((mpt = TRAN2MPT(tran)) == NULL)
1718         return (DDI_SUCCESS);
1720     /*
1721      * Send RAID action system shutdown to sync IR.  Disable HBA
1722      * interrupts in hardware first.
1723      */
1724     MPTSAS_DISABLE_INTR(mpt);
1725     mptsas_raid_action_system_shutdown(mpt);
1727     return (DDI_SUCCESS);
1728 }
1729 #else /* __sparc */
1730 /*
1731  * quiesce(9E) entry point.
1732  *
1733  * This function is called when the system is single-threaded at high
1734  * PIL with preemption disabled. Therefore, this function must not be
1735  * blocked.
1736  *
1737  * This function returns DDI_SUCCESS on success, or DDI_FAILURE on failure.
1738  * DDI_FAILURE indicates an error condition and should almost never happen.
1739  */
1740 static int
1741 mptsas_quiesce(dev_info_t *devi)
1742 {

```

```

1743     mptsas_t      *mpt;
1744     scsi_hba_tran_t *tran;
1746     /*
1747      * If this call is for iport, just return.
1748      */
1749     if (scsi_hba_iport_unit_address(devi))
1750         return (DDI_SUCCESS);
1752     if ((tran = ddi_get_driver_private(devi)) == NULL)
1753         return (DDI_SUCCESS);
1755     if ((mpt = TRAN2MPT(tran)) == NULL)
1756         return (DDI_SUCCESS);
1758     /* Disable HBA interrupts in hardware */
1759     MPTSAS_DISABLE_INTR(mpt);
1760     /* Send RAID action system shutdown to sync IR */
1761     mptsas_raid_action_system_shutdown(mpt);
1763     return (DDI_SUCCESS);
1764 }
1765 #endif /* __sparc */
1767 /*
1768  * detach(9E).  Remove all device allocations and system resources;
1769  * disable device interrupts.
1770  * Return DDI_SUCCESS if done; DDI_FAILURE if there's a problem.
1771  */
1772 static int
1773 mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
1774 {
1775     /* CONSTCOND */
1776     ASSERT(NO_COMPETING_THREADS);
1777     NDBG0(("mptsas_detach: dip=0x%p cmd=0x%p", (void *)devi, (void *)cmd));
1779     switch (cmd) {
1780     case DDI_DETACH:
1781         return (mptsas_do_detach(devi));
1783     case DDI_SUSPEND:
1784         return (mptsas_suspend(devi));
1786     default:
1787         return (DDI_FAILURE);
1788     }
1789     /* NOTREACHED */
1790 }
1792 static int
1793 mptsas_do_detach(dev_info_t *dip)
1794 {
1795     mptsas_t      *mpt;
1796     scsi_hba_tran_t *tran;
1797     int            circ = 0;
1798     int            circ1 = 0;
1799     mdi_pathinfo_t *pip = NULL;
1800     int            i;
1801     int            doneq_thread_num = 0;
1803     NDBG0(("mptsas_do_detach: dip=0x%p", (void *)dip));
1805     if ((tran = ndi_flavorv_get(dip, SCSA_FLAVOR_SCSI_DEVICE)) == NULL)
1806         return (DDI_FAILURE);
1808     mpt = TRAN2MPT(tran);

```

```

1809     if (!mpt) {
1810         return (DDI_FAILURE);
1811     }
1812     /*
1813      * Still have pathinfo child, should not detach mpt driver
1814      */
1815     if (scsi_hba_iport_unit_address(dip)) {
1816         if (mpt->m_mpxio_enable) {
1817             /*
1818              * MPxIO enabled for the iport
1819              */
1820             ndi_devi_enter(scsi_vhci_dip, &circ1);
1821             ndi_devi_enter(dip, &circ);
1822             while (pip = mdi_get_next_client_path(dip, NULL)) {
1823                 if (mdi_pi_free(pip, 0) == MDI_SUCCESS) {
1824                     continue;
1825                 }
1826                 ndi_devi_exit(dip, circ);
1827                 ndi_devi_exit(scsi_vhci_dip, circ1);
1828                 NDBG12(("detach failed because of "
1829                     "outstanding path info"));
1830                 return (DDI_FAILURE);
1831             }
1832             ndi_devi_exit(dip, circ);
1833             ndi_devi_exit(scsi_vhci_dip, circ1);
1834             (void) mdi_phci_unregister(dip, 0);
1835         }
1837         ddi_prop_remove_all(dip);
1839         return (DDI_SUCCESS);
1840     }
1842     /* Make sure power level is D0 before accessing registers */
1843     if (mpt->m_options & MPTSAS_OPT_PM) {
1844         (void) pm_busy_component(dip, 0);
1845         if (mpt->m_power_level != PM_LEVEL_D0) {
1846             if (pm_raise_power(dip, 0, PM_LEVEL_D0) !=
1847                 DDI_SUCCESS) {
1848                 mptsas_log(mpt, CE_WARN,
1849                     "mptsas%d: Raise power request failed.",
1850                     mpt->m_instance);
1851                 (void) pm_idle_component(dip, 0);
1852                 return (DDI_FAILURE);
1853             }
1854         }
1855     }
1857     /*
1858      * Send RAID action system shutdown to sync IR. After action, send a
1859      * Message Unit Reset. Since after that DMA resource will be freed,
1860      * set ioc to READY state will avoid HBA initiated DMA operation.
1861      */
1862     mutex_enter(&mpt->m_mutex);
1863     MPTSAS_DISABLE_INTR(mpt);
1864     mptsas_raid_action_system_shutdown(mpt);
1865     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1866     (void) mptsas_ioc_reset(mpt, FALSE);
1867     mutex_exit(&mpt->m_mutex);
1868     mptsas_rem_intrs(mpt);
1869     ddi_taskq_destroy(mpt->m_event_taskq);
1870     ddi_taskq_destroy(mpt->m_dr_taskq);
1872     if (mpt->m_doneq_thread_n) {
1873         mutex_enter(&mpt->m_doneq_mutex);
1874         doneq_thread_num = mpt->m_doneq_thread_n;

```

```

1875         for (i = 0; i < mpt->m_doneq_thread_n; i++) {
1876             mutex_enter(&mpt->m_doneq_thread_id[i].mutex);
1877             mpt->m_doneq_thread_id[i].flag &=
1878                 (~MPTSAS_DONEQ_THREAD_ACTIVE);
1879             cv_signal(&mpt->m_doneq_thread_id[i].cv);
1880             mutex_exit(&mpt->m_doneq_thread_id[i].mutex);
1881         }
1882         while (mpt->m_doneq_thread_n) {
1883             cv_wait(&mpt->m_doneq_thread_cv,
1884                 &mpt->m_doneq_mutex);
1885         }
1886         for (i = 0; i < doneq_thread_num; i++) {
1887             cv_destroy(&mpt->m_doneq_thread_id[i].cv);
1888             mutex_destroy(&mpt->m_doneq_thread_id[i].mutex);
1889         }
1890         kmem_free(mpt->m_doneq_thread_id,
1891             sizeof(mptsas_doneq_thread_list_t)
1892                 * doneq_thread_num);
1893         mutex_exit(&mpt->m_doneq_mutex);
1894         cv_destroy(&mpt->m_doneq_thread_cv);
1895         mutex_destroy(&mpt->m_doneq_mutex);
1896     }
1898     scsi_hba_reset_notify_tear_down(mpt->m_reset_notify_listf);
1900     mptsas_list_del(mpt);
1902     /*
1903      * Cancel timeout threads for this mpt
1904      */
1905     mutex_enter(&mpt->m_mutex);
1906     if (mpt->m_quiesce_timeid) {
1907         timeout_id_t tid = mpt->m_quiesce_timeid;
1908         mpt->m_quiesce_timeid = 0;
1909         mutex_exit(&mpt->m_mutex);
1910         (void) untimeout(tid);
1911         mutex_enter(&mpt->m_mutex);
1912     }
1914     if (mpt->m_restart_cmd_timeid) {
1915         timeout_id_t tid = mpt->m_restart_cmd_timeid;
1916         mpt->m_restart_cmd_timeid = 0;
1917         mutex_exit(&mpt->m_mutex);
1918         (void) untimeout(tid);
1919         mutex_enter(&mpt->m_mutex);
1920     }
1922     mutex_exit(&mpt->m_mutex);
1924     /*
1925      * last mpt? ... if active, CANCEL watch threads.
1926      */
1927     mutex_enter(&mptsas_global_mutex);
1928     if (mptsas_head == NULL) {
1929         timeout_id_t tid;
1930         /*
1931          * Clear mptsas_timeouts_enable so that the watch thread
1932          * gets restarted on DDI_ATTACH
1933          */
1934         mptsas_timeouts_enabled = 0;
1935         if (mptsas_timeout_id) {
1936             tid = mptsas_timeout_id;
1937             mptsas_timeout_id = 0;
1938             mutex_exit(&mptsas_global_mutex);
1939             (void) untimeout(tid);
1940             mutex_enter(&mptsas_global_mutex);

```

```

1941     }
1942     if (mptsas_reset_watch) {
1943         tid = mptsas_reset_watch;
1944         mptsas_reset_watch = 0;
1945         mutex_exit(&mptsas_global_mutex);
1946         (void) untimeout(tid);
1947         mutex_enter(&mptsas_global_mutex);
1948     }
1949 }
1950 mutex_exit(&mptsas_global_mutex);
1951
1952 /*
1953  * Delete Phy stats
1954  */
1955 mptsas_destroy_phy_stats(mpt);
1956
1957 mptsas_destroy_hashes(mpt);
1958
1959 /*
1960  * Delete nt_active.
1961  */
1962 mutex_enter(&mpt->m_mutex);
1963 mptsas_free_active_slots(mpt);
1964 mutex_exit(&mpt->m_mutex);
1965
1966 /* deallocate everything that was allocated in mptsas_attach */
1967 mptsas_cache_destroy(mpt);
1968
1969 mptsas_hba_fini(mpt);
1970 mptsas_cfg_fini(mpt);
1971
1972 /* Lower the power informing PM Framework */
1973 if (mpt->m_options & MPTSAS_OPT_PM) {
1974     if (pm_lower_power(dip, 0, PM_LEVEL_D3) != DDI_SUCCESS)
1975         mptsas_log(mpt, CE_WARN,
1976             "!mptsas%d: Lower power request failed "
1977             "during detach, ignoring.",
1978             mpt->m_instance);
1979 }
1980
1981 mutex_destroy(&mpt->m_tx_waitq_mutex);
1982 mutex_destroy(&mpt->m_passthru_mutex);
1983 mutex_destroy(&mpt->m_mutex);
1984 for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1985     mutex_destroy(&mpt->m_phy_info[i].smhba_info.phy_mutex);
1986 }
1987 cv_destroy(&mpt->m_cv);
1988 cv_destroy(&mpt->m_passthru_cv);
1989 cv_destroy(&mpt->m_fw_cv);
1990 cv_destroy(&mpt->m_config_cv);
1991 cv_destroy(&mpt->m_fw_diag_cv);
1992
1993
1994 mptsas_smp_teardown(mpt);
1995 mptsas_hba_teardown(mpt);
1996
1997 mptsas_config_space_fini(mpt);
1998
1999 mptsas_free_handshake_msg(mpt);
2000
2001 mptsas_fm_fini(mpt);
2002 ddi_soft_state_free(mptsas_state, ddi_get_instance(dip));
2003 ddi_prop_remove_all(dip);
2004
2005 return (DDI_SUCCESS);
2006 }

```

```

2008 static void
2009 mptsas_list_add(mptsas_t *mpt)
2010 {
2011     rw_enter(&mptsas_global_rwlock, RW_WRITER);
2012
2013     if (mptsas_head == NULL) {
2014         mptsas_head = mpt;
2015     } else {
2016         mptsas_tail->m_next = mpt;
2017     }
2018     mptsas_tail = mpt;
2019     rw_exit(&mptsas_global_rwlock);
2020 }
2021
2022 static void
2023 mptsas_list_del(mptsas_t *mpt)
2024 {
2025     mptsas_t *m;
2026     /*
2027      * Remove device instance from the global linked list
2028      */
2029     rw_enter(&mptsas_global_rwlock, RW_WRITER);
2030     if (mptsas_head == mpt) {
2031         m = mptsas_head = mpt->m_next;
2032     } else {
2033         for (m = mptsas_head; m != NULL; m = m->m_next) {
2034             if (m->m_next == mpt) {
2035                 m->m_next = mpt->m_next;
2036                 break;
2037             }
2038         }
2039         if (m == NULL) {
2040             mptsas_log(mpt, CE_PANIC, "Not in softc list!");
2041         }
2042     }
2043
2044     if (mptsas_tail == mpt) {
2045         mptsas_tail = m;
2046     }
2047     rw_exit(&mptsas_global_rwlock);
2048 }
2049
2050 static int
2051 mptsas_alloc_handshake_msg(mptsas_t *mpt, size_t alloc_size)
2052 {
2053     ddi_dma_attr_t task_dma_attrs;
2054
2055     mpt->m_hshk_dma_size = 0;
2056     task_dma_attrs = mpt->m_msg_dma_attr;
2057     task_dma_attrs.dma_attr_sgllen = 1;
2058     task_dma_attrs.dma_attr_granular = (uint32_t)(alloc_size);
2059
2060     /* allocate Task Management ddi_dma resources */
2061     if (mptsas_dma_addr_create(mpt, task_dma_attrs,
2062         &mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl, &mpt->m_hshk_memp,
2063         alloc_size, NULL) == FALSE) {
2064         return (DDI_FAILURE);
2065     }
2066     mpt->m_hshk_dma_size = alloc_size;
2067
2068     return (DDI_SUCCESS);
2069 }
2070
2071 static void
2072 mptsas_free_handshake_msg(mptsas_t *mpt)

```



```

2073 {
2074     if (mpt->m_hshk_dma_size == 0)
2075         return;
2076     mptsas_dma_addr_destroy(&mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl);
2077     mpt->m_hshk_dma_size = 0;
2078 }

2080 static int
2081 mptsas_hba_setup(mptsas_t *mpt)
2082 {
2083     scsi_hba_tran_t      *hba_tran;
2084     int                  tran_flags;

2086     /* Allocate a transport structure */
2087     hba_tran = mpt->m_tran = scsi_hba_tran_alloc(mpt->m_dip,
2088         SCSI_HBA_CANSLEEP);
2089     ASSERT(mpt->m_tran != NULL);

2091     hba_tran->tran_hba_private = mpt;
2092     hba_tran->tran_tgt_private = NULL;

2094     hba_tran->tran_tgt_init = mptsas_scsi_tgt_init;
2095     hba_tran->tran_tgt_free = mptsas_scsi_tgt_free;

2097     hba_tran->tran_start = mptsas_scsi_start;
2098     hba_tran->tran_reset = mptsas_scsi_reset;
2099     hba_tran->tran_abort = mptsas_scsi_abort;
2100     hba_tran->tran_getcap = mptsas_scsi_getcap;
2101     hba_tran->tran_setcap = mptsas_scsi_setcap;
2102     hba_tran->tran_init_pkt = mptsas_scsi_init_pkt;
2103     hba_tran->tran_destroy_pkt = mptsas_scsi_destroy_pkt;

2105     hba_tran->tran_dmafree = mptsas_scsi_dmafree;
2106     hba_tran->tran_sync_pkt = mptsas_scsi_sync_pkt;
2107     hba_tran->tran_reset_notify = mptsas_scsi_reset_notify;

2109     hba_tran->tran_get_bus_addr = mptsas_get_bus_addr;
2110     hba_tran->tran_get_name = mptsas_get_name;

2112     hba_tran->tran_quiesce = mptsas_scsi_quiesce;
2113     hba_tran->tran_unquiesce = mptsas_scsi_unquiesce;
2114     hba_tran->tran_bus_reset = NULL;

2116     hba_tran->tran_add_eventcall = NULL;
2117     hba_tran->tran_get_eventcookie = NULL;
2118     hba_tran->tran_post_event = NULL;
2119     hba_tran->tran_remove_eventcall = NULL;

2121     hba_tran->tran_bus_config = mptsas_bus_config;

2123     hba_tran->tran_interconnect_type = INTERCONNECT_SAS;

2125     /*
2126     * All children of the HBA are iports. We need tran was cloned.
2127     * So we pass the flags to SCSI. SCSI_HBA_TRAN_CLONE will be
2128     * inherited to iport's tran vector.
2129     */
2130     tran_flags = (SCSI_HBA_HBA | SCSI_HBA_TRAN_CLONE);

2132     if (scsi_hba_attach_setup(mpt->m_dip, &mpt->m_msg_dma_attr,
2133         hba_tran, tran_flags) != DDI_SUCCESS) {
2134         mptsas_log(mpt, CE_WARN, "hba attach setup failed");
2135         scsi_hba_tran_free(hba_tran);
2136         mpt->m_tran = NULL;
2137         return (FALSE);
2138     }

```

```

2139         return (TRUE);
2140     }

2142 static void
2143 mptsas_hba_teardown(mptsas_t *mpt)
2144 {
2145     (void) scsi_hba_detach(mpt->m_dip);
2146     if (mpt->m_tran != NULL) {
2147         scsi_hba_tran_free(mpt->m_tran);
2148         mpt->m_tran = NULL;
2149     }
2150 }

2152 static void
2153 mptsas_iport_register(mptsas_t *mpt)
2154 {
2155     int i, j;
2156     mptsas_phymask_t      mask = 0x0;
2157     /*
2158     * initial value of mask is 0
2159     */
2160     mutex_enter(&mpt->m_mutex);
2161     for (i = 0; i < mpt->m_num_phys; i++) {
2162         mptsas_phymask_t phy_mask = 0x0;
2163         char phy_mask_name[MPTSAS_MAX_PHYS];
2164         uint8_t current_port;

2166         if (mpt->m_phy_info[i].attached_devhdl == 0)
2167             continue;

2169         bzero(phy_mask_name, sizeof(phy_mask_name));

2171         current_port = mpt->m_phy_info[i].port_num;

2173         if ((mask & (1 << i)) != 0)
2174             continue;

2176         for (j = 0; j < mpt->m_num_phys; j++) {
2177             if (mpt->m_phy_info[j].attached_devhdl &&
2178                 (mpt->m_phy_info[j].port_num == current_port)) {
2179                 phy_mask |= (1 << j);
2180             }
2181         }
2182         mask = mask | phy_mask;

2184         for (j = 0; j < mpt->m_num_phys; j++) {
2185             if ((phy_mask >> j) & 0x01) {
2186                 mpt->m_phy_info[j].phy_mask = phy_mask;
2187             }
2188         }

2190         (void) sprintf(phy_mask_name, "%x", phy_mask);

2192         mutex_exit(&mpt->m_mutex);
2193         /*
2194         * register a iport
2195         */
2196         (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
2197         mutex_enter(&mpt->m_mutex);
2198     }
2199     mutex_exit(&mpt->m_mutex);
2200     /*
2201     * register a virtual port for RAID volume always
2202     */
2203     (void) scsi_hba_iport_register(mpt->m_dip, "v0");

```

```

2205 }
2207 static int
2208 mptsas_smp_setup(mptsas_t *mpt)
2209 {
2210     mpt->m_smptran = smp_hba_tran_alloc(mpt->m_dip);
2211     ASSERT(mpt->m_smptran != NULL);
2212     mpt->m_smptran->smp_tran_hba_private = mpt;
2213     mpt->m_smptran->smp_tran_start = mptsas_smp_start;
2214     if (smp_hba_attach_setup(mpt->m_dip, mpt->m_smptran) != DDI_SUCCESS) {
2215         mptsas_log(mpt, CE_WARN, "smp attach setup failed");
2216         smp_hba_tran_free(mpt->m_smptran);
2217         mpt->m_smptran = NULL;
2218         return (FALSE);
2219     }
2220     /*
2221     * Initialize smp hash table
2222     */
2223     mpt->m_smp_target = rehash_create(MPTSAS_SMP_BUCKET_COUNT,
2224     mptsas_target_addr_hash, mptsas_target_addr_cmp,
2225     mptsas_smp_free, sizeof (mptsas_smp_t),
2226     offsetof(mptsas_smp_t, m_link), offsetof(mptsas_smp_t, m_addr),
2227     KM_SLEEP);
2228     mpt->m_smp_devhdl = 0xFFFF;
2230     return (TRUE);
2231 }
2233 static void
2234 mptsas_smp_teardown(mptsas_t *mpt)
2235 {
2236     (void) smp_hba_detach(mpt->m_dip);
2237     if (mpt->m_smptran != NULL) {
2238         smp_hba_tran_free(mpt->m_smptran);
2239         mpt->m_smptran = NULL;
2240     }
2241     mpt->m_smp_devhdl = 0;
2242 }
2244 static int
2245 mptsas_cache_create(mptsas_t *mpt)
2246 {
2247     int instance = mpt->m_instance;
2248     char buf[64];
2250     /*
2251     * create kmem cache for packets
2252     */
2253     (void) sprintf(buf, "mptsas%d_cache", instance);
2254     mpt->m_kmem_cache = kmem_cache_create(buf,
2255     sizeof (struct mptsas_cmd) + scsi_pkt_size(), 8,
2256     mptsas_kmem_cache_constructor, mptsas_kmem_cache_destructor,
2257     NULL, (void *)mpt, NULL, 0);
2259     if (mpt->m_kmem_cache == NULL) {
2260         mptsas_log(mpt, CE_WARN, "creating kmem cache failed");
2261         return (FALSE);
2262     }
2264     /*
2265     * create kmem cache for extra SGL frames if SGL cannot
2266     * be accommodated into main request frame.
2267     */
2268     (void) sprintf(buf, "mptsas%d_cache_frames", instance);
2269     mpt->m_cache_frames = kmem_cache_create(buf,
2270     sizeof (mptsas_cache_frames_t), 8,

```

```

2271     mptsas_cache_frames_constructor, mptsas_cache_frames_destructor,
2272     NULL, (void *)mpt, NULL, 0);
2274     if (mpt->m_cache_frames == NULL) {
2275         mptsas_log(mpt, CE_WARN, "creating cache for frames failed");
2276         return (FALSE);
2277     }
2279     return (TRUE);
2280 }
2282 static void
2283 mptsas_cache_destroy(mptsas_t *mpt)
2284 {
2285     /* deallocate in reverse order */
2286     if (mpt->m_cache_frames) {
2287         kmem_cache_destroy(mpt->m_cache_frames);
2288         mpt->m_cache_frames = NULL;
2289     }
2290     if (mpt->m_kmem_cache) {
2291         kmem_cache_destroy(mpt->m_kmem_cache);
2292         mpt->m_kmem_cache = NULL;
2293     }
2294 }
2296 static int
2297 mptsas_power(dev_info_t *dip, int component, int level)
2298 {
2299     #ifndef __lock_lint
2300         _NOTE(ARGUNUSED(component))
2301     #endif
2302     mptsas_t *mpt;
2303     int rval = DDI_SUCCESS;
2304     int polls = 0;
2305     uint32_t ioc_status;
2307     if (scsi_hba_iport_unit_address(dip) != 0)
2308         return (DDI_SUCCESS);
2310     mpt = ddi_get_soft_state(mptsas_state, ddi_get_instance(dip));
2311     if (mpt == NULL) {
2312         return (DDI_FAILURE);
2313     }
2315     mutex_enter(&mpt->m_mutex);
2317     /*
2318     * If the device is busy, don't lower its power level
2319     */
2320     if (mpt->m_busy && (mpt->m_power_level > level)) {
2321         mutex_exit(&mpt->m_mutex);
2322         return (DDI_FAILURE);
2323     }
2324     switch (level) {
2325     case PM_LEVEL_D0:
2326         NDBG11(("mptsas%d: turning power ON.", mpt->m_instance));
2327         MPTSAS_POWER_ON(mpt);
2328         /*
2329         * Wait up to 30 seconds for IOC to come out of reset.
2330         */
2331         while (((ioc_status = ddi_get32(mpt->m_datap,
2332         &mpt->m_reg->Doorbell)) &
2333         MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_RESET) {
2334             if (polls++ > 3000) {
2335                 break;
2336             }

```

```

2337         delay(drv_usecstohz(10000));
2338     }
2339     /*
2340     * If IOC is not in operational state, try to hard reset it.
2341     */
2342     if ((ioc_status & MPI2_IOC_STATE_MASK) !=
2343         MPI2_IOC_STATE_OPERATIONAL) {
2344         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
2345         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
2346             mptsas_log(mpt, CE_WARN,
2347                 "mptsas power: hard reset failed");
2348             mutex_exit(&mpt->m_mutex);
2349             return (DDI_FAILURE);
2350         }
2351     }
2352     mpt->m_power_level = PM_LEVEL_D0;
2353     break;
2354 case PM_LEVEL_D3:
2355     NDBG11(("mptsas%d: turning power OFF.", mpt->m_instance));
2356     MPTSAS_POWER_OFF(mpt);
2357     break;
2358 default:
2359     mptsas_log(mpt, CE_WARN, "mptsas%d: unknown power level <%x>.",
2360         mpt->m_instance, level);
2361     rval = DDI_FAILURE;
2362     break;
2363 }
2364 mutex_exit(&mpt->m_mutex);
2365 return (rval);
2366 }

2368 /*
2369 * Initialize configuration space and figure out which
2370 * chip and revision of the chip the mpt driver is using.
2371 */
2372 static int
2373 mptsas_config_space_init(mptsas_t *mpt)
2374 {
2375     NDBG0(("mptsas_config_space_init"));

2377     if (mpt->m_config_handle != NULL)
2378         return (TRUE);

2380     if (pci_config_setup(mpt->m_dip,
2381         &mpt->m_config_handle) != DDI_SUCCESS) {
2382         mptsas_log(mpt, CE_WARN, "cannot map configuration space.");
2383         return (FALSE);
2384     }

2386     /*
2387     * This is a workaround for a XMIT5 ASIC bug which does not
2388     * drive the CBE upper bits.
2389     */
2390     if (pci_config_get16(mpt->m_config_handle, PCI_CONF_STAT) &
2391         PCI_STAT_PERROR) {
2392         pci_config_put16(mpt->m_config_handle, PCI_CONF_STAT,
2393             PCI_STAT_PERROR);
2394     }

2396     mptsas_setup_cmd_reg(mpt);

2398     /*
2399     * Get the chip device id:
2400     */
2401     mpt->m_devid = pci_config_get16(mpt->m_config_handle, PCI_CONF_DEVID);

```

```

2403     /*
2404     * Save the revision.
2405     */
2406     mpt->m_revid = pci_config_get8(mpt->m_config_handle, PCI_CONF_REVID);

2408     /*
2409     * Save the SubSystem Vendor and Device IDs
2410     */
2411     mpt->m_svid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBVENID);
2412     mpt->m_ssid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBSYSID);

2414     /*
2415     * Set the latency timer to 0x40 as specified by the upa -> pci
2416     * bridge chip design team. This may be done by the sparc pci
2417     * bus nexus driver, but the driver should make sure the latency
2418     * timer is correct for performance reasons.
2419     */
2420     pci_config_put8(mpt->m_config_handle, PCI_CONF_LATENCY_TIMER,
2421         MPTSAS_LATENCY_TIMER);

2423     (void) mptsas_get_pci_cap(mpt);
2424     return (TRUE);
2425 }

2427 static void
2428 mptsas_config_space_fini(mptsas_t *mpt)
2429 {
2430     if (mpt->m_config_handle != NULL) {
2431         mptsas_disable_bus_master(mpt);
2432         pci_config_teardown(&mpt->m_config_handle);
2433         mpt->m_config_handle = NULL;
2434     }
2435 }

2437 static void
2438 mptsas_setup_cmd_reg(mptsas_t *mpt)
2439 {
2440     ushort_t    cmdreg;

2442     /*
2443     * Set the command register to the needed values.
2444     */
2445     cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
2446     cmdreg |= (PCI_COMM_ME | PCI_COMM_SERR_ENABLE |
2447         PCI_COMM_PARITY_DETECT | PCI_COMM_MAE);
2448     cmdreg &= ~PCI_COMM_IO;
2449     pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
2450 }

2452 static void
2453 mptsas_disable_bus_master(mptsas_t *mpt)
2454 {
2455     ushort_t    cmdreg;

2457     /*
2458     * Clear the master enable bit in the PCI command register.
2459     * This prevents any bus mastering activity like DMA.
2460     */
2461     cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
2462     cmdreg &= ~PCI_COMM_ME;
2463     pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
2464 }

2466 int
2467 mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_statep)
2468 {

```

```

2469     ddi_dma_attr_t  attrs;

2471     attrs = mpt->m_io_dma_attr;
2472     attrs.dma_attr_sgllen = 1;

2474     ASSERT(dma_statep != NULL);

2476     if (mptsas_dma_addr_create(mpt, attrs, &dma_statep->handle,
2477         &dma_statep->accessp, &dma_statep->memp, dma_statep->size,
2478         &dma_statep->cookie) == FALSE) {
2479         return (DDI_FAILURE);
2480     }

2482     return (DDI_SUCCESS);
2483 }

2485 void
2486 mptsas_dma_free(mptsas_dma_alloc_state_t *dma_statep)
2487 {
2488     ASSERT(dma_statep != NULL);
2489     mptsas_dma_addr_destroy(&dma_statep->handle, &dma_statep->accessp);
2490     dma_statep->size = 0;
2491 }

2493 int
2494 mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)())
2495 {
2496     ddi_dma_attr_t      attrs;
2497     ddi_dma_handle_t    dma_handle;
2498     caddr_t             memp;
2499     ddi_acc_handle_t    accessp;
2500     int                 rval;

2502     ASSERT(mutex_owned(&mpt->m_mutex));

2504     attrs = mpt->m_msg_dma_attr;
2505     attrs.dma_attr_sgllen = 1;
2506     attrs.dma_attr_granular = size;

2508     if (mptsas_dma_addr_create(mpt, attrs, &dma_handle,
2509         &accessp, &memp, size, NULL) == FALSE) {
2510         return (DDI_FAILURE);
2511     }

2513     rval = (*callback) (mpt, memp, var, accessp);

2515     if ((mptsas_check_dma_handle(dma_handle) != DDI_SUCCESS) ||
2516         (mptsas_check_acc_handle(accessp) != DDI_SUCCESS)) {
2517         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2518         rval = DDI_FAILURE;
2519     }

2521     mptsas_dma_addr_destroy(&dma_handle, &accessp);
2522     return (rval);

2524 }

2526 static int
2527 mptsas_alloc_request_frames(mptsas_t *mpt)
2528 {
2529     ddi_dma_attr_t      frame_dma_attrs;
2530     caddr_t             memp;
2531     ddi_dma_cookie_t    cookie;
2532     size_t              mem_size;

2534     /*

```

```

2535     * re-alloc when it has already allocated
2536     */
2537     mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
2538         &mpt->m_acc_req_frame_hdl);

2540     /*
2541     * The size of the request frame pool is:
2542     *   Number of Request Frames * Request Frame Size
2543     */
2544     mem_size = mpt->m_max_requests * mpt->m_req_frame_size;

2546     /*
2547     * set the DMA attributes. System Request Message Frames must be
2548     * aligned on a 16-byte boundary.
2549     */
2550     frame_dma_attrs = mpt->m_msg_dma_attr;
2551     frame_dma_attrs.dma_attr_align = 16;
2552     frame_dma_attrs.dma_attr_sgllen = 1;

2554     /*
2555     * allocate the request frame pool.
2556     */
2557     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2558         &mpt->m_dma_req_frame_hdl, &mpt->m_acc_req_frame_hdl, &memp,
2559         mem_size, &cookie) == FALSE) {
2560         return (DDI_FAILURE);
2561     }

2563     /*
2564     * Store the request frame memory address. This chip uses this
2565     * address to dma to and from the driver's frame. The second
2566     * address is the address mpt uses to fill in the frame.
2567     */
2568     mpt->m_req_frame_dma_addr = cookie.dmac_laddress;
2569     mpt->m_req_frame = memp;

2571     /*
2572     * Clear the request frame pool.
2573     */
2574     bzero(mpt->m_req_frame, mem_size);

2576     return (DDI_SUCCESS);
2577 }

2579 static int
2580 mptsas_alloc_reply_frames(mptsas_t *mpt)
2581 {
2582     ddi_dma_attr_t      frame_dma_attrs;
2583     caddr_t             memp;
2584     ddi_dma_cookie_t    cookie;
2585     size_t              mem_size;

2587     /*
2588     * re-alloc when it has already allocated
2589     */
2590     mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
2591         &mpt->m_acc_reply_frame_hdl);

2593     /*
2594     * The size of the reply frame pool is:
2595     *   Number of Reply Frames * Reply Frame Size
2596     */
2597     mem_size = mpt->m_max_replies * mpt->m_reply_frame_size;

2599     /*
2600     * set the DMA attributes. System Reply Message Frames must be

```

```

2601     * aligned on a 4-byte boundry. This is the default.
2602     */
2603     frame_dma_attrs = mpt->m_msg_dma_attr;
2604     frame_dma_attrs.dma_attr_sgllen = 1;
2605
2606     /*
2607     * allocate the reply frame pool
2608     */
2609     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2610         &mpt->m_dma_reply_frame_hdl, &mpt->m_acc_reply_frame_hdl, &memp,
2611         mem_size, &cookie) == FALSE) {
2612         return (DDI_FAILURE);
2613     }
2614
2615     /*
2616     * Store the reply frame memory address. This chip uses this
2617     * address to dma to and from the driver's frame. The second
2618     * address is the address mpt uses to process the frame.
2619     */
2620     mpt->m_reply_frame_dma_addr = cookie.dmac_address;
2621     mpt->m_reply_frame = memp;
2622
2623     /*
2624     * Clear the reply frame pool.
2625     */
2626     bzero(mpt->m_reply_frame, mem_size);
2627
2628     return (DDI_SUCCESS);
2629 }
2630
2631 static int
2632 mptsas_alloc_free_queue(mptsas_t *mpt)
2633 {
2634     ddi_dma_attr_t      frame_dma_attrs;
2635     caddr_t             memp;
2636     ddi_dma_cookie_t    cookie;
2637     size_t              mem_size;
2638
2639     /*
2640     * re-alloc when it has already allocated
2641     */
2642     mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
2643         &mpt->m_acc_free_queue_hdl);
2644
2645     /*
2646     * The reply free queue size is:
2647     * Reply Free Queue Depth * 4
2648     * The "4" is the size of one 32 bit address (low part of 64-bit
2649     * address)
2650     */
2651     mem_size = mpt->m_free_queue_depth * 4;
2652
2653     /*
2654     * set the DMA attributes The Reply Free Queue must be aligned on a
2655     * 16-byte boundry.
2656     */
2657     frame_dma_attrs = mpt->m_msg_dma_attr;
2658     frame_dma_attrs.dma_attr_align = 16;
2659     frame_dma_attrs.dma_attr_sgllen = 1;
2660
2661     /*
2662     * allocate the reply free queue
2663     */
2664     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2665         &mpt->m_dma_free_queue_hdl, &mpt->m_acc_free_queue_hdl, &memp,
2666         mem_size, &cookie) == FALSE) {

```

```

2667         return (DDI_FAILURE);
2668     }
2669
2670     /*
2671     * Store the reply free queue memory address. This chip uses this
2672     * address to read from the reply free queue. The second address
2673     * is the address mpt uses to manage the queue.
2674     */
2675     mpt->m_free_queue_dma_addr = cookie.dmac_address;
2676     mpt->m_free_queue = memp;
2677
2678     /*
2679     * Clear the reply free queue memory.
2680     */
2681     bzero(mpt->m_free_queue, mem_size);
2682
2683     return (DDI_SUCCESS);
2684 }
2685
2686 static int
2687 mptsas_alloc_post_queue(mptsas_t *mpt)
2688 {
2689     ddi_dma_attr_t      frame_dma_attrs;
2690     caddr_t             memp;
2691     ddi_dma_cookie_t    cookie;
2692     size_t              mem_size;
2693
2694     /*
2695     * re-alloc when it has already allocated
2696     */
2697     mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
2698         &mpt->m_acc_post_queue_hdl);
2699
2700     /*
2701     * The reply descriptor post queue size is:
2702     * Reply Descriptor Post Queue Depth * 8
2703     * The "8" is the size of each descriptor (8 bytes or 64 bits).
2704     */
2705     mem_size = mpt->m_post_queue_depth * 8;
2706
2707     /*
2708     * set the DMA attributes. The Reply Descriptor Post Queue must be
2709     * aligned on a 16-byte boundry.
2710     */
2711     frame_dma_attrs = mpt->m_msg_dma_attr;
2712     frame_dma_attrs.dma_attr_align = 16;
2713     frame_dma_attrs.dma_attr_sgllen = 1;
2714
2715     /*
2716     * allocate the reply post queue
2717     */
2718     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2719         &mpt->m_dma_post_queue_hdl, &mpt->m_acc_post_queue_hdl, &memp,
2720         mem_size, &cookie) == FALSE) {
2721         return (DDI_FAILURE);
2722     }
2723
2724     /*
2725     * Store the reply descriptor post queue memory address. This chip
2726     * uses this address to write to the reply descriptor post queue. The
2727     * second address is the address mpt uses to manage the queue.
2728     */
2729     mpt->m_post_queue_dma_addr = cookie.dmac_address;
2730     mpt->m_post_queue = memp;
2731
2732     /*

```

```

2733     * Clear the reply post queue memory.
2734     */
2735     bzero(mpt->m_post_queue, mem_size);

2737     return (DDI_SUCCESS);
2738 }

2740 static void
2741 mptsas_alloc_reply_args(mptsas_t *mpt)
2742 {
2743     if (mpt->m_replyh_args == NULL) {
2744         mpt->m_replyh_args = kmem_zalloc(sizeof (m_replyh_arg_t) *
2745             mpt->m_max_replies, KM_SLEEP);
2746     }
2747 }

2749 static int
2750 mptsas_alloc_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)
2751 {
2752     mptsas_cache_frames_t *frames = NULL;
2753     if (cmd->cmd_extra_frames == NULL) {
2754         frames = kmem_cache_alloc(mpt->m_cache_frames, KM_NOSLEEP);
2755         if (frames == NULL) {
2756             return (DDI_FAILURE);
2757         }
2758         cmd->cmd_extra_frames = frames;
2759     }
2760     return (DDI_SUCCESS);
2761 }

2763 static void
2764 mptsas_free_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)
2765 {
2766     if (cmd->cmd_extra_frames) {
2767         kmem_cache_free(mpt->m_cache_frames,
2768             (void *)cmd->cmd_extra_frames);
2769         cmd->cmd_extra_frames = NULL;
2770     }
2771 }

2773 static void
2774 mptsas_cfg_fini(mptsas_t *mpt)
2775 {
2776     NDBG0(("mptsas_cfg_fini"));
2777     ddi_regs_map_free(&mpt->m_datap);
2778 }

2780 static void
2781 mptsas_hba_fini(mptsas_t *mpt)
2782 {
2783     NDBG0(("mptsas_hba_fini"));

2785     /*
2786     * Free up any allocated memory
2787     */
2788     mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
2789         &mpt->m_acc_req_frame_hdl);

2791     mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
2792         &mpt->m_acc_reply_frame_hdl);

2794     mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
2795         &mpt->m_acc_free_queue_hdl);

2797     mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
2798         &mpt->m_acc_post_queue_hdl);

```

```

2800     if (mpt->m_replyh_args != NULL) {
2801         kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
2802             * mpt->m_max_replies);
2803     }
2804 }

2806 static int
2807 mptsas_name_child(dev_info_t *lun_dip, char *name, int len)
2808 {
2809     int         lun = 0;
2810     char        *sas_wwn = NULL;
2811     int         phynum = -1;
2812     int         reallen = 0;

2814     /* Get the target num */
2815     lun = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip, DDI_PROP_DONTPASS,
2816         LUN_PROP, 0);

2818     if ((phynum = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip,
2819         DDI_PROP_DONTPASS, "sata-phy", -1)) != -1) {
2820         /*
2821         * Stick in the address of form "pPHY,LUN"
2822         */
2823         reallen = snprintf(name, len, "p%x,%x", phynum, lun);
2824     } else if (ddi_prop_lookup_string(DDI_DEV_T_ANY, lun_dip,
2825         DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &sas_wwn)
2826         == DDI_PROP_SUCCESS) {
2827         /*
2828         * Stick in the address of the form "wwWN,LUN"
2829         */
2830         reallen = snprintf(name, len, "%s,%x", sas_wwn, lun);
2831         ddi_prop_free(sas_wwn);
2832     } else {
2833         return (DDI_FAILURE);
2834     }

2836     ASSERT(reallen < len);
2837     if (reallen >= len) {
2838         mptsas_log(0, CE_WARN, "!mptsas_get_name: name parameter "
2839             "length too small, it needs to be %d bytes", reallen + 1);
2840     }
2841     return (DDI_SUCCESS);
2842 }

2844 /*
2845 * tran_tgt_init(9E) - target device instance initialization
2846 */
2847 static int
2848 mptsas_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
2849     scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
2850 {
2851     #ifndef __lock_lint
2852         _NOTE(ARGUNUSED(hba_tran))
2853     #endif

2855     /*
2856     * At this point, the scsi_device structure already exists
2857     * and has been initialized.
2858     *
2859     * Use this function to allocate target-private data structures,
2860     * if needed by this HBA. Add revised flow-control and queue
2861     * properties for child here, if desired and if you can tell they
2862     * support tagged queueing by now.
2863     */
2864     mptsas_t         *mpt;

```

```

2865     int                lun = sd->sd_address.a_lun;
2866     mdi_pathinfo_t     *pip = NULL;
2867     mptsas_tgt_private_t *tgt_private = NULL;
2868     mptsas_target_t    *tgt = NULL;
2869     char               *psas_wwn = NULL;
2870     mptsas_phymask_t   phymask = 0;
2871     uint64_t           sas_wwn = 0;
2872     mptsas_target_addr_t addr;
2873     mpt = SDEV2MPT(sd);

2875     ASSERT(scsi_hba_iport_unit_address(hba_dip) != 0);

2877     NDBG0(("mptsas_scsi_tgt_init: hbadip=0x%p tgt_dip=0x%p lun=%d",
2878         (void *)hba_dip, (void *)tgt_dip, lun));

2880     if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
2881         (void) ndi_merge_node(tgt_dip, mptsas_name_child);
2882         ddi_set_name_addr(tgt_dip, NULL);
2883         return (DDI_FAILURE);
2884     }
2885     /*
2886     * phymask is 0 means the virtual port for RAID
2887     */
2888     phymask = (mptsas_phymask_t) ddi_prop_get_int(DDI_DEV_T_ANY, hba_dip, 0,
2889         "phymask", 0);
2890     if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
2891         if ((pip = (void *) (sd->sd_private)) == NULL) {
2892             /*
2893             * Very bad news if this occurs. Somehow scsi_vhci has
2894             * lost the pathinfo node for this target.
2895             */
2896             return (DDI_NOT_WELL_FORMED);
2897         }

2899         if (mdi_prop_lookup_int(pip, LUN_PROP, &lun) !=
2900             DDI_PROP_SUCCESS) {
2901             mptsas_log(mpt, CE_WARN, "Get lun property failed\n");
2902             return (DDI_FAILURE);
2903         }

2905         if (mdi_prop_lookup_string(pip, SCSI_ADDR_PROP_TARGET_PORT,
2906             &psas_wwn) == MDI_SUCCESS) {
2907             if (scsi_wwnstr_to_wwn(psas_wwn, &sas_wwn)) {
2908                 sas_wwn = 0;
2909             }
2910             (void) mdi_prop_free(psas_wwn);
2911         }
2912     } else {
2913         lun = ddi_prop_get_int(DDI_DEV_T_ANY, tgt_dip,
2914             DDI_PROP_DONTPASS, LUN_PROP, 0);
2915         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, tgt_dip,
2916             DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &psas_wwn) ==
2917             DDI_PROP_SUCCESS) {
2918             if (scsi_wwnstr_to_wwn(psas_wwn, &sas_wwn)) {
2919                 sas_wwn = 0;
2920             }
2921             ddi_prop_free(psas_wwn);
2922         } else {
2923             sas_wwn = 0;
2924         }
2925     }

2927     ASSERT((sas_wwn != 0) || (phymask != 0));
2928     addr.mta_wwn = sas_wwn;
2929     addr.mta_phymask = phymask;
2930     mutex_enter(&mpt->m_mutex);

```

```

2931     ptgt = rehash_lookup(mpt->m_targets, &addr);
2932     mutex_exit(&mpt->m_mutex);
2933     if (ptgt == NULL) {
2934         mptsas_log(mpt, CE_WARN, "!tgt_init: target doesn't exist or "
2935             "gone already! phymask:%x, saswwn %"PRIx64, phymask,
2936             sas_wwn);
2937         return (DDI_FAILURE);
2938     }
2939     if (hba_tran->tran_tgt_private == NULL) {
2940         tgt_private = kmem_zalloc(sizeof (mptsas_tgt_private_t),
2941             KM_SLEEP);
2942         tgt_private->t_lun = lun;
2943         tgt_private->t_private = ptgt;
2944         hba_tran->tran_tgt_private = tgt_private;
2945     }

2947     if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
2948         return (DDI_SUCCESS);
2949     }
2950     mutex_enter(&mpt->m_mutex);

2952     if (ptgt->m_deviceinfo &
2953         (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
2954             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
2955         uchar_t *inq89 = NULL;
2956         int inq89_len = 0x238;
2957         int reallen = 0;
2958         int rval = 0;
2959         struct sata_id *sid = NULL;
2960         char model[SATA_ID_MODEL_LEN + 1];
2961         char fw[SATA_ID_FW_LEN + 1];
2962         char *vid, *pid;
2963         int i;

2965         mutex_exit(&mpt->m_mutex);
2966         /*
2967         * According SCSI/ATA Translation -2 (SAT-2) revision 01a
2968         * chapter 12.4.2 VPD page 89h includes 512 bytes ATA IDENTIFY
2969         * DEVICE data or ATA IDENTIFY PACKET DEVICE data.
2970         */
2971         inq89 = kmem_zalloc(inq89_len, KM_SLEEP);
2972         rval = mptsas_inquiry(mpt, ptgt, 0, 0x89,
2973             inq89, inq89_len, &reallen, 1);

2975         if (rval != 0) {
2976             if (inq89 != NULL) {
2977                 kmem_free(inq89, inq89_len);
2978             }

2980             mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
2981                 "0x89 for SATA target:%x failed!", ptgt->m_devhdl);
2982             return (DDI_SUCCESS);
2983         }
2984         sid = (void *) (&inq89[60]);

2986         swab(sid->ai_model, model, SATA_ID_MODEL_LEN);
2987         swab(sid->ai_fw, fw, SATA_ID_FW_LEN);

2989         model[SATA_ID_MODEL_LEN] = 0;
2990         fw[SATA_ID_FW_LEN] = 0;

2992         /*
2993         * split model into into vid/pid
2994         */
2995         for (i = 0, pid = model; i < SATA_ID_MODEL_LEN; i++, pid++)
2996             if ((*pid == ' ') || (*pid == '\t'))

```

```

2997         break;
2998     if (i < SATA_ID_MODEL_LEN) {
2999         vid = model;
3000         /*
3001          * terminate vid, establish pid
3002          */
3003         *pid++ = 0;
3004     } else {
3005         /*
3006          * vid will stay "ATA", the rule is same
3007          * as sata framework implementation.
3008          */
3009         vid = NULL;
3010         /*
3011          * model is all pid
3012          */
3013         pid = model;
3014     }
3015
3016     /*
3017     * override SCSI "inquiry-*" properties
3018     */
3019     if (vid)
3020         (void) scsi_device_prop_update_inqstring(sd,
3021         INQUIRY_VENDOR_ID, vid, strlen(vid));
3022     if (pid)
3023         (void) scsi_device_prop_update_inqstring(sd,
3024         INQUIRY_PRODUCT_ID, pid, strlen(pid));
3025     (void) scsi_device_prop_update_inqstring(sd,
3026     INQUIRY_REVISION_ID, fw, strlen(fw));
3027
3028     if (inq89 != NULL) {
3029         kmem_free(inq89, inq89_len);
3030     }
3031     } else {
3032         mutex_exit(&mpt->m_mutex);
3033     }
3034
3035     return (DDI_SUCCESS);
3036 }
3037 /*
3038  * tran_tgt_free(9E) - target device instance deallocation
3039  */
3040 static void
3041 mptsas_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
3042     scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
3043 {
3044     #ifndef __lock_lint
3045         _NOTE(ARGUNUSED(hba_dip, tgt_dip, hba_tran, sd))
3046     #endif
3047
3048     mptsas_tgt_private_t *tgt_private = hba_tran->tran_tgt_private;
3049
3050     if (tgt_private != NULL) {
3051         kmem_free(tgt_private, sizeof (mptsas_tgt_private_t));
3052         hba_tran->tran_tgt_private = NULL;
3053     }
3054 }
3055
3056 /*
3057  * scsi_pkt handling
3058  *
3059  * Visible to the external world via the transport structure.
3060  */
3061
3062 /*

```

```

3063  * Notes:
3064  * - transport the command to the addressed SCSI target/lun device
3065  * - normal operation is to schedule the command to be transported,
3066  *   and return TRAN_ACCEPT if this is successful.
3067  * - if NO_INTR, tran_start must poll device for command completion
3068  */
3069 static int
3070 mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
3071 {
3072     #ifndef __lock_lint
3073         _NOTE(ARGUNUSED(ap))
3074     #endif
3075     mptsas_t *mpt = PKT2MPT(pkt);
3076     mptsas_cmd_t *cmd = PKT2CMD(pkt);
3077     int rval;
3078     mptsas_target_t *tgt = cmd->cmd_tgt_addr;
3079
3080     NDBG1(("mptsas_scsi_start: pkt=0x%p", (void *)pkt));
3081     ASSERT(tgt);
3082     if (tgt == NULL)
3083         return (TRAN_FATAL_ERROR);
3084
3085     /*
3086     * prepare the pkt before taking mutex.
3087     */
3088     rval = mptsas_prepare_pkt(cmd);
3089     if (rval != TRAN_ACCEPT) {
3090         return (rval);
3091     }
3092
3093     /*
3094     * Send the command to target/lun, however your HBA requires it.
3095     * If busy, return TRAN_BUSY; if there's some other formatting error
3096     * in the packet, return TRAN_BADPKT; otherwise, fall through to the
3097     * return of TRAN_ACCEPT.
3098     *
3099     * Remember that access to shared resources, including the mptsas_t
3100     * data structure and the HBA hardware registers, must be protected
3101     * with mutexes, here and everywhere.
3102     *
3103     * Also remember that at interrupt time, you'll get an argument
3104     * to the interrupt handler which is a pointer to your mptsas_t
3105     * structure; you'll have to remember which commands are outstanding
3106     * and which scsi_pkt is the currently-running command so the
3107     * interrupt handler can refer to the pkt to set completion
3108     * status, call the target driver back through pkt_comp, etc.
3109     *
3110     * If the instance lock is held by other thread, don't spin to wait
3111     * for it. Instead, queue the cmd and next time when the instance lock
3112     * is not held, accept all the queued cmd. A extra tx_waitq is
3113     * introduced to protect the queue.
3114     *
3115     * The polled cmd will not be queued and accepted as usual.
3116     *
3117     * Under the tx_waitq mutex, record whether a thread is draining
3118     * the tx_waitq. An IO requesting thread that finds the instance
3119     * mutex contended appends to the tx_waitq and while holding the
3120     * tx_wait mutex, if the draining flag is not set, sets it and then
3121     * proceeds to spin for the instance mutex. This scheme ensures that
3122     * the last cmd in a burst be processed.
3123     *
3124     * we enable this feature only when the helper threads are enabled,
3125     * at which we think the loads are heavy.
3126     *
3127     * per instance mutex m_tx_waitq_mutex is introduced to protect the
3128     * m_tx_waitqtail, m_tx_waitq, m_tx_draining.

```



```

3129      */
3131      if (mpt->m_doneq_thread_n) {
3132          if (mutex_tryenter(&mpt->m_mutex) != 0) {
3133              rval = mptsas_accept_txdq_and_pkt(mpt, cmd);
3134              mutex_exit(&mpt->m_mutex);
3135          } else if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3136              mutex_enter(&mpt->m_mutex);
3137              rval = mptsas_accept_txdq_and_pkt(mpt, cmd);
3138              mutex_exit(&mpt->m_mutex);
3139          } else {
3140              mutex_enter(&mpt->m_tx_waitq_mutex);
3141              /*
3142               * ptgt->m_dr_flag is protected by m_mutex or
3143               * m_tx_waitq_mutex. In this case, m_tx_waitq_mutex
3144               * is acquired.
3145               */
3146              if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3147                  if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3148                      /*
3149                       * The command should be allowed to
3150                       * retry by returning TRAN_BUSY to
3151                       * stall the I/O's which come from
3152                       * scsi_vhci since the device/path is
3153                       * in unstable state now.
3154                       */
3155                      mutex_exit(&mpt->m_tx_waitq_mutex);
3156                      return (TRAN_BUSY);
3157                  } else {
3158                      /*
3159                       * The device is offline, just fail the
3160                       * command by returning
3161                       * TRAN_FATAL_ERROR.
3162                       */
3163                      mutex_exit(&mpt->m_tx_waitq_mutex);
3164                      return (TRAN_FATAL_ERROR);
3165                  }
3166              }
3167              if (mpt->m_tx_draining) {
3168                  cmd->cmd_flags |= CFLAG_TXQ;
3169                  *mpt->m_tx_waitqtail = cmd;
3170                  mpt->m_tx_waitqtail = &cmd->cmd_linkp;
3171                  mutex_exit(&mpt->m_tx_waitq_mutex);
3172              } else { /* drain the queue */
3173                  mpt->m_tx_draining = 1;
3174                  mutex_exit(&mpt->m_tx_waitq_mutex);
3175                  mutex_enter(&mpt->m_mutex);
3176                  rval = mptsas_accept_txdq_and_pkt(mpt, cmd);
3177                  mutex_exit(&mpt->m_mutex);
3178              }
3179          }
3180      } else {
3181          mutex_enter(&mpt->m_mutex);
3182          /*
3183           * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
3184           * in this case, m_mutex is acquired.
3185           */
3186          if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3187              if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3188                  /*
3189                   * commands should be allowed to retry by
3190                   * returning TRAN_BUSY to stall the I/O's
3191                   * which come from scsi_vhci since the device/
3192                   * path is in unstable state now.
3193                   */
3194                  mutex_exit(&mpt->m_mutex);

```

```

3195          return (TRAN_BUSY);
3196      } else {
3197          /*
3198           * The device is offline, just fail the
3199           * command by returning TRAN_FATAL_ERROR.
3200           */
3201          mutex_exit(&mpt->m_mutex);
3202          return (TRAN_FATAL_ERROR);
3203      }
3204  }
3205  rval = mptsas_accept_pkt(mpt, cmd);
3206  mutex_exit(&mpt->m_mutex);
3207  }
3209  return (rval);
3210 }
3212 /*
3213  * Accept all the queued cmds(if any) before accept the current one.
3214  */
3215 static int
3216 mptsas_accept_txdq_and_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3217 {
3218     int rval;
3219     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
3221     ASSERT(mutex_owned(&mpt->m_mutex));
3222     /*
3223      * The call to mptsas_accept_tx_waitq() must always be performed
3224      * because that is where mpt->m_tx_draining is cleared.
3225      */
3226     mutex_enter(&mpt->m_tx_waitq_mutex);
3227     mptsas_accept_tx_waitq(mpt);
3228     mutex_exit(&mpt->m_tx_waitq_mutex);
3229     /*
3230      * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
3231      * in this case, m_mutex is acquired.
3232      */
3233     if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3234         if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3235             /*
3236              * The command should be allowed to retry by returning
3237              * TRAN_BUSY to stall the I/O's which come from
3238              * scsi_vhci since the device/path is in unstable state
3239              * now.
3240              */
3241             return (TRAN_BUSY);
3242         } else {
3243             /*
3244              * The device is offline, just fail the command by
3245              * return TRAN_FATAL_ERROR.
3246              */
3247             return (TRAN_FATAL_ERROR);
3248         }
3249     }
3250     rval = mptsas_accept_pkt(mpt, cmd);
3252     return (rval);
3253 }
3255 static int
3256 mptsas_accept_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3257 {
3258     int        rval = TRAN_ACCEPT;
3259     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

```

```

3261     NDBG1(("mptsas_accept_pkt: cmd=0x%p", (void *)cmd));
3263     ASSERT(mutex_owed(&mpt->m_mutex));

3265     if ((cmd->cmd_flags & CFLAG_PREPARED) == 0) {
3266         rval = mptsas_prepare_pkt(cmd);
3267         if (rval != TRAN_ACCEPT) {
3268             cmd->cmd_flags &= ~CFLAG_TRANFLAG;
3269             return (rval);
3270         }
3271     }

3273     /*
3274     * reset the throttle if we were draining
3275     */
3276     if ((ptgt->m_t_ncmds == 0) &&
3277         (ptgt->m_t_throttle == DRAIN_THROTTLE)) {
3278         NDBG23(("reset throttle"));
3279         ASSERT(ptgt->m_reset_delay == 0);
3280         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
3281     }

3283     /*
3284     * If HBA is being reset, the DevHandles are being re-initialized,
3285     * which means that they could be invalid even if the target is still
3286     * attached. Check if being reset and if DevHandle is being
3287     * re-initialized. If this is the case, return BUSY so the I/O can be
3288     * retried later.
3289     */
3290     if ((ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) && mpt->m_in_reset) {
3291         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
3292         if (cmd->cmd_flags & CFLAG_TXQ) {
3293             mptsas_doneq_add(mpt, cmd);
3294             mptsas_doneq_empty(mpt);
3295             return (rval);
3296         } else {
3297             return (TRAN_BUSY);
3298         }
3299     }

3301     /*
3302     * If device handle has already been invalidated, just
3303     * fail the command. In theory, command from scsi_vhci
3304     * client is impossible send down command with invalid
3305     * devhdl since devhdl is set after path offline, target
3306     * driver is not suppose to select a offlined path.
3307     */
3308     if (ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) {
3309         NDBG3(("rejecting command, it might because invalid devhdl "
3310             "request."));
3311         mptsas_set_pkt_reason(mpt, cmd, CMD_DEV_GONE, STAT_TERMINATED);
3312         if (cmd->cmd_flags & CFLAG_TXQ) {
3313             mptsas_doneq_add(mpt, cmd);
3314             mptsas_doneq_empty(mpt);
3315             return (rval);
3316         } else {
3317             return (TRAN_FATAL_ERROR);
3318         }
3319     }
3320     /*
3321     * The first case is the normal case. mpt gets a command from the
3322     * target driver and starts it.
3323     * Since SMID 0 is reserved and the TM slot is reserved, the actual max
3324     * commands is m_max_requests - 2.
3325     */
3326     if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&

```

```

3327         (ptgt->m_t_throttle > HOLD_THROTTLE) &&
3328         (ptgt->m_t_ncmds < ptgt->m_t_throttle) &&
3329         (ptgt->m_reset_delay == 0) &&
3330         (ptgt->m_t_nwait == 0) &&
3331         ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0)) {
3332             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
3333                 (void) mptsas_start_cmd(mpt, cmd);
3334             } else {
3335                 mptsas_waitq_add(mpt, cmd);
3336             }
3337         } else {
3338             /*
3339             * Add this pkt to the work queue
3340             */
3341             mptsas_waitq_add(mpt, cmd);

3343             if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3344                 (void) mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);

3346                 /*
3347                 * Only flush the doneq if this is not a TM
3348                 * cmd. For TM cmds the flushing of the
3349                 * doneq will be done in those routines.
3350                 */
3351                 if ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
3352                     mptsas_doneq_empty(mpt);
3353                 }
3354             }
3355         }
3356         return (rval);
3357     }

3359     int
3360     mptsas_save_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
3361     {
3362         mptsas_slots_t *slots = mpt->m_active;
3363         uint_t slot, start_rotor;
3364         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

3366         ASSERT(MUTEX_HELD(&mpt->m_mutex));

3368         /*
3369         * Account for reserved TM request slot and reserved SMID of 0.
3370         */
3371         ASSERT(slots->m_n_normal == (mpt->m_max_requests - 2));

3373         /*
3374         * Find the next available slot, beginning at m_rotor. If no slot is
3375         * available, we'll return FALSE to indicate that. This mechanism
3376         * considers only the normal slots, not the reserved slot 0 nor the
3377         * task management slot m_n_normal + 1. The rotor is left to point to
3378         * the normal slot after the one we select, unless we select the last
3379         * normal slot in which case it returns to slot 1.
3380         */
3381         start_rotor = slots->m_rotor;
3382         do {
3383             slot = slots->m_rotor++;
3384             if (slots->m_rotor > slots->m_n_normal)
3385                 slots->m_rotor = 1;

3387             if (slots->m_rotor == start_rotor)
3388                 break;
3389         } while (slots->m_slot[slot] != NULL);

3391         if (slots->m_slot[slot] != NULL)
3392             return (FALSE);

```

```

3394     ASSERT(slot != 0 && slot <= slots->m_n_normal);
3396     cmd->cmd_slot = slot;
3397     slots->m_slot[slot] = cmd;
3398     mpt->m_ncmds++;
3400     /*
3401     * only increment per target ncmds if this is not a
3402     * command that has no target associated with it (i.e. a
3403     * event acknowledgment)
3404     */
3405     if ((cmd->cmd_flags & CFLAG_CMDIOCB) == 0) {
3406         /*
3407         * Expiration time is set in mptsas_start_cmd
3408         */
3409         ptgt->m_t_ncmds++;
3410         cmd->cmd_active_expiration = 0;
3411     } else {
3412         /*
3413         * Initialize expiration time for passthrough commands,
3414         */
3415         cmd->cmd_active_expiration = gethrtime() +
3416             (hrtime_t)cmd->cmd_pkt->pkt_time * NANOSEC;
3417     }
3418     return (TRUE);
3419 }
3421 /*
3422 * prepare the pkt:
3423 * the pkt may have been resubmitted or just reused so
3424 * initialize some fields and do some checks.
3425 */
3426 static int
3427 mptsas_prepare_pkt(mptsas_cmd_t *cmd)
3428 {
3429     struct scsi_pkt *pkt = CMD2PKT(cmd);
3431     NDBG1(("mptsas_prepare_pkt: cmd=0x%p", (void *)cmd));
3433     /*
3434     * Reinitialize some fields that need it; the packet may
3435     * have been resubmitted
3436     */
3437     pkt->pkt_reason = CMD_CMPLT;
3438     pkt->pkt_state = 0;
3439     pkt->pkt_statistics = 0;
3440     pkt->pkt_resid = 0;
3441     cmd->cmd_age = 0;
3442     cmd->cmd_pkt_flags = pkt->pkt_flags;
3444     /*
3445     * zero status byte.
3446     */
3447     *(pkt->pkt_scbp) = 0;
3449     if (cmd->cmd_flags & CFLAG_DMAVALID) {
3450         pkt->pkt_resid = cmd->cmd_dmaaccount;
3452         /*
3453         * consistent packets need to be sync'ed first
3454         * (only for data going out)
3455         */
3456         if ((cmd->cmd_flags & CFLAG_CMDIOCB) &&
3457             (cmd->cmd_flags & CFLAG_DMASEND)) {
3458             (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,

```

```

3459         DDI_DMA_SYNC_FORDEV);
3460     }
3461 }
3463     cmd->cmd_flags =
3464         (cmd->cmd_flags & ~(CFLAG_TRANFLAG)) |
3465         CFLAG_PREPARED | CFLAG_IN_TRANSPORT;
3467     return (TRAN_ACCEPT);
3468 }
3470 /*
3471 * tran_init_pkt(9E) - allocate scsi_pkt(9S) for command
3472 *
3473 * One of three possibilities:
3474 * - allocate scsi_pkt
3475 * - allocate scsi_pkt and DMA resources
3476 * - allocate DMA resources to an already-allocated pkt
3477 */
3478 static struct scsi_pkt *
3479 mptsas_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
3480     struct buf *bp, int cmdlen, int statuslen, int tgtlen, int flags,
3481     int (*callback)(), caddr_t arg)
3482 {
3483     mptsas_cmd_t *cmd, *new_cmd;
3484     mptsas_t *mpt = ADDR2MPT(ap);
3485     int failure = 1;
3486     uint_t oldcookie;
3487     mptsas_target_t *ptgt = NULL;
3488     int rval;
3489     mptsas_tgt_private_t *tgt_private;
3490     int kf;
3492     kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;
3494     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
3495         tran_tgt_private;
3496     ASSERT(tgt_private != NULL);
3497     if (tgt_private == NULL) {
3498         return (NULL);
3499     }
3500     ptgt = tgt_private->t_private;
3501     ASSERT(ptgt != NULL);
3502     if (ptgt == NULL)
3503         return (NULL);
3504     ap->a_target = ptgt->m_devhdl;
3505     ap->a_lun = tgt_private->t_lun;
3507     ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);
3508 #ifdef MPTSAS_TEST_EXTRN_ALLOC
3509     statuslen *= 100; tgtlen *= 4;
3510 #endif
3511     NDBG3(("mptsas_scsi_init_pkt:\n"
3512         "\ttgt=%d in=0x%p bp=0x%p clen=%d slen=%d tlen=%d flags=%x",
3513         ap->a_target, (void *)pkt, (void *)bp,
3514         cmdlen, statuslen, tgtlen, flags));
3516     /*
3517     * Allocate the new packet.
3518     */
3519     if (pkt == NULL) {
3520         ddi_dma_handle_t save_dma_handle;
3521         ddi_dma_handle_t save_arq_dma_handle;
3522         struct buf *save_arq_bp;
3523         ddi_dma_cookie_t save_arqcookie;

```

```

3525     cmd = kmem_cache_alloc(mpt->m_kmem_cache, kf);
3527     if (cmd) {
3528         save_dma_handle = cmd->cmd_dmahandle;
3529         save_arq_dma_handle = cmd->cmd_arqhandle;
3530         save_arq_bp = cmd->cmd_arq_buf;
3531         save_arqcookie = cmd->cmd_arqcookie;
3532         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3533         cmd->cmd_dmahandle = save_dma_handle;
3534         cmd->cmd_arqhandle = save_arq_dma_handle;
3535         cmd->cmd_arq_buf = save_arq_bp;
3536         cmd->cmd_arqcookie = save_arqcookie;
3538
3539         pkt = (void *)((uchar_t *)cmd +
3540             sizeof (struct mptsas_cmd));
3541         pkt->pkt_ha_private = (opaque_t)cmd;
3542         pkt->pkt_address = *ap;
3543         pkt->pkt_private = (opaque_t)cmd->cmd_pkt_private;
3544         pkt->pkt_scbp = (opaque_t)&cmd->cmd_scb;
3545         pkt->pkt_cdbp = (opaque_t)&cmd->cmd_cdb;
3546         cmd->cmd_pkt = (struct scsi_pkt *)pkt;
3547         cmd->cmd_cdblen = (uchar_t)cmdlen;
3548         cmd->cmd_scblen = statuslen;
3549         cmd->cmd_rqslen = SENSE_LENGTH;
3550         cmd->cmd_tgt_addr = ptgt;
3551         failure = 0;
3553     }
3554     if (failure || (cmdlen > sizeof (cmd->cmd_cdb)) ||
3555         (tgtlen > PKT_PRIV_LEN) ||
3556         (statuslen > EXTCMDSTATUS_SIZE)) {
3557         if (failure == 0) {
3558             /*
3559              * if extern alloc fails, all will be
3560              * deallocated, including cmd
3561              */
3562             failure = mptsas_pkt_alloc_extern(mpt, cmd,
3563                 cmdlen, tgtlen, statuslen, kf);
3564         }
3565         if (failure) {
3566             /*
3567              * if extern allocation fails, it will
3568              * deallocate the new pkt as well
3569              */
3570             return (NULL);
3571         }
3572         new_cmd = cmd;
3574     } else {
3575         cmd = PKT2CMD(pkt);
3576         new_cmd = NULL;
3577     }
3580     /* grab cmd->cmd_cookiec here as oldcookiec */
3582     oldcookiec = cmd->cmd_cookiec;
3584     /*
3585     * If the dma was broken up into PARTIAL transfers cmd_nwin will be
3586     * greater than 0 and we'll need to grab the next dma window
3587     */
3588     /*
3589     * SLM-not doing extra command frame right now; may add later
3590     */

```

```

3592     if (cmd->cmd_nwin > 0) {
3594         /*
3595         * Make sure we haven't gone past the the total number
3596         * of windows
3597         */
3598         if (++cmd->cmd_winindex >= cmd->cmd_nwin) {
3599             return (NULL);
3600         }
3601         if (ddi_dma_getwin(cmd->cmd_dmahandle, cmd->cmd_winindex,
3602             &cmd->cmd_dma_offset, &cmd->cmd_dma_len,
3603             &cmd->cmd_cookie, &cmd->cmd_cookiec) == DDI_FAILURE) {
3604             return (NULL);
3605         }
3606         goto get_dma_cookies;
3607     }
3610     if (flags & PKT_XARQ) {
3611         cmd->cmd_flags |= CFLAG_XARQ;
3612     }
3614     /*
3615     * DMA resource allocation. This version assumes your
3616     * HBA has some sort of bus-mastering or onboard DMA capability, with a
3617     * scatter-gather list of length MPTSAS_MAX_DMA_SEGS, as given in the
3618     * ddi_dma_attr_t structure and passed to scsi_impl_dmaget.
3619     */
3620     if (bp && (bp->b_bcount != 0) &&
3621         (cmd->cmd_flags & CFLAG_DMAVALID) == 0) {
3623         int     cnt, dma_flags;
3624         mptti_t *dmap; /* ptr to the S/G list */
3626         /*
3627         * Set up DMA memory and position to the next DMA segment.
3628         */
3629         ASSERT(cmd->cmd_dmahandle != NULL);
3631         if (bp->b_flags & B_READ) {
3632             dma_flags = DDI_DMA_READ;
3633             cmd->cmd_flags &= ~CFLAG_DMASEND;
3634         } else {
3635             dma_flags = DDI_DMA_WRITE;
3636             cmd->cmd_flags |= CFLAG_DMASEND;
3637         }
3638         if (flags & PKT_CONSISTENT) {
3639             cmd->cmd_flags |= CFLAG_CMDIOPB;
3640             dma_flags |= DDI_DMA_CONSISTENT;
3641         }
3643         if (flags & PKT_DMA_PARTIAL) {
3644             dma_flags |= DDI_DMA_PARTIAL;
3645         }
3647         /*
3648         * workaround for byte hole issue on psycho and
3649         * schizo pre 2.1
3650         */
3651         if ((bp->b_flags & B_READ) && ((bp->b_flags &
3652             (B_PAGEIO|B_REMAPPED)) != B_PAGEIO) &&
3653             ((uintptr_t)bp->b_un.b_addr & 0x7)) {
3654             dma_flags |= DDI_DMA_CONSISTENT;
3655         }

```

```

3657     rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
3658     dma_flags, callback, arg,
3659     &cmd->cmd_cookie, &cmd->cmd_cookiec);
3660     if (rval == DDI_DMA_PARTIAL_MAP) {
3661         (void) ddi_dma_numwin(cmd->cmd_dmahandle,
3662         &cmd->cmd_nwin);
3663         cmd->cmd_winindex = 0;
3664         (void) ddi_dma_getwin(cmd->cmd_dmahandle,
3665         cmd->cmd_winindex, &cmd->cmd_dma_offset,
3666         &cmd->cmd_dma_len, &cmd->cmd_cookie,
3667         &cmd->cmd_cookiec);
3668     } else if (rval && (rval != DDI_DMA_MAPPED)) {
3669         switch (rval) {
3670         case DDI_DMA_NORESOURCES:
3671             bioerror(bp, 0);
3672             break;
3673         case DDI_DMA_BADATTR:
3674         case DDI_DMA_NOMAPPING:
3675             bioerror(bp, EFAULT);
3676             break;
3677         case DDI_DMA_TOOBIG:
3678         default:
3679             bioerror(bp, EINVAL);
3680             break;
3681         }
3682         cmd->cmd_flags &= ~CFLAG_DMAVALID;
3683         if (new_cmd) {
3684             mptsas_scsi_destroy_pkt(ap, pkt);
3685         }
3686         return ((struct scsi_pkt *)NULL);
3687     }
3689 get_dma_cookies:
3690     cmd->cmd_flags |= CFLAG_DMAVALID;
3691     ASSERT(cmd->cmd_cookiec > 0);
3693     if (cmd->cmd_cookiec > MPTSAS_MAX_CMD_SEGS) {
3694         mptsas_log(mpt, CE_NOTE, "large cookiec received %d\n",
3695         cmd->cmd_cookiec);
3696         bioerror(bp, EINVAL);
3697         if (new_cmd) {
3698             mptsas_scsi_destroy_pkt(ap, pkt);
3699         }
3700         return ((struct scsi_pkt *)NULL);
3701     }
3703     /*
3704     * Allocate extra SGL buffer if needed.
3705     */
3706     if ((cmd->cmd_cookiec > MPTSAS_MAX_FRAME_SGES64(mpt)) &&
3707     (cmd->cmd_extra_frames == NULL)) {
3708         if (mptsas_alloc_extra_sgl_frame(mpt, cmd) ==
3709         DDI_FAILURE) {
3710             mptsas_log(mpt, CE_WARN, "MPT SGL mem alloc "
3711             "failed");
3712             bioerror(bp, ENOMEM);
3713             if (new_cmd) {
3714                 mptsas_scsi_destroy_pkt(ap, pkt);
3715             }
3716             return ((struct scsi_pkt *)NULL);
3717         }
3718     }
3720     /*
3721     * Always use scatter-gather transfer
3722     * Use the loop below to store physical addresses of

```

```

3723     * DMA segments, from the DMA cookies, into your HBA's
3724     * scatter-gather list.
3725     * We need to ensure we have enough kmem alloc'd
3726     * for the sg entries since we are no longer using an
3727     * array inside mptsas_cmd_t.
3728     *
3729     * We check cmd->cmd_cookiec against oldcookiec so
3730     * the scatter-gather list is correctly allocated
3731     */
3733     if (oldcookiec != cmd->cmd_cookiec) {
3734         if (cmd->cmd_sg != (mptti_t *)NULL) {
3735             kmem_free(cmd->cmd_sg, sizeof (mptti_t) *
3736             oldcookiec);
3737             cmd->cmd_sg = NULL;
3738         }
3739     }
3741     if (cmd->cmd_sg == (mptti_t *)NULL) {
3742         cmd->cmd_sg = kmem_alloc((size_t)(sizeof (mptti_t)*
3743         cmd->cmd_cookiec), kf);
3745         if (cmd->cmd_sg == (mptti_t *)NULL) {
3746             mptsas_log(mpt, CE_WARN,
3747             "unable to kmem_alloc enough memory "
3748             "for scatter/gather list");
3749         }
3750         /*
3751         * if we have an ENOMEM condition we need to behave
3752         * the same way as the rest of this routine
3753         */
3754         bioerror(bp, ENOMEM);
3755         if (new_cmd) {
3756             mptsas_scsi_destroy_pkt(ap, pkt);
3757         }
3758         return ((struct scsi_pkt *)NULL);
3759     }
3760 }
3762     dmap = cmd->cmd_sg;
3764     ASSERT(cmd->cmd_cookie.dmac_size != 0);
3766     /*
3767     * store the first segment into the S/G list
3768     */
3769     dmap->count = cmd->cmd_cookie.dmac_size;
3770     dmap->addr.address64.Low = (uint32_t)
3771     (cmd->cmd_cookie.dmac_laddress & 0xffffffffull);
3772     dmap->addr.address64.High = (uint32_t)
3773     (cmd->cmd_cookie.dmac_laddress >> 32);
3775     /*
3776     * dmacount counts the size of the dma for this window
3777     * (if partial dma is being used). totaldmacount
3778     * keeps track of the total amount of dma we have
3779     * transferred for all the windows (needed to calculate
3780     * the resid value below).
3781     */
3782     cmd->cmd_dmacount = cmd->cmd_cookie.dmac_size;
3783     cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;
3785     /*
3786     * We already stored the first DMA scatter gather segment,
3787     * start at 1 if we need to store more.
3788     */

```

```

3789     for (cnt = 1; cnt < cmd->cmd_cookiec; cnt++) {
3790         /*
3791          * Get next DMA cookie
3792          */
3793         ddi_dma_nextcookie(cmd->cmd_dmahandle,
3794             &cmd->cmd_cookie);
3795         dmap++;

3797         cmd->cmd_dmacount += cmd->cmd_cookie.dmac_size;
3798         cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;

3800         /*
3801          * store the segment parms into the S/G list
3802          */
3803         dmap->count = cmd->cmd_cookie.dmac_size;
3804         dmap->addr.address64.Low = (uint32_t)
3805             (cmd->cmd_cookie.dmac_laddress & 0xfffffffffull);
3806         dmap->addr.address64.High = (uint32_t)
3807             (cmd->cmd_cookie.dmac_laddress >> 32);
3808     }

3810     /*
3811     * If this was partially allocated we set the resid
3812     * the amount of data NOT transferred in this window
3813     * If there is only one window, the resid will be 0
3814     */
3815     pkt->pkt_resid = (bp->b_bcount - cmd->cmd_totaldmacount);
3816     NDBG3(("mptsas_scsi_init_pkt: cmd_dmacount=%d.",
3817         cmd->cmd_dmacount));
3818 }
3819 return (pkt);
3820 }

3822 /*
3823 * tran_destroy_pkt(9E) - scsi_pkt(9s) deallocation
3824 *
3825 * Notes:
3826 * - also frees DMA resources if allocated
3827 * - implicit DMA synchronization
3828 */
3829 static void
3830 mptsas_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
3831 {
3832     mptsas_cmd_t *cmd = PKT2CMD(pkt);
3833     mptsas_t *mpt = ADDR2MPT(ap);

3835     NDBG3(("mptsas_scsi_destroy_pkt: target=%d pkt=0x%p",
3836         ap->a_target, (void *)pkt));

3838     if (cmd->cmd_flags & CFLAG_DMAVALID) {
3839         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
3840         cmd->cmd_flags &= ~CFLAG_DMAVALID;
3841     }

3843     if (cmd->cmd_sg) {
3844         kmem_free(cmd->cmd_sg, sizeof (mptti_t) * cmd->cmd_cookiec);
3845         cmd->cmd_sg = NULL;
3846     }

3848     mptsas_free_extra_sgl_frame(mpt, cmd);

3850     if ((cmd->cmd_flags &
3851         (CFLAG_FREE | CFLAG_CDBEXTERN | CFLAG_PRIVEXTERN |
3852         CFLAG_SCBEXTERN)) == 0) {
3853         cmd->cmd_flags = CFLAG_FREE;
3854         kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);

```

```

3855     } else {
3856         mptsas_pkt_destroy_extern(mpt, cmd);
3857     }
3858 }

3860 /*
3861 * kmem cache constructor and destructor:
3862 * When constructing, we bzero the cmd and allocate the dma handle
3863 * When destructing, just free the dma handle
3864 */
3865 static int
3866 mptsas_kmem_cache_constructor(void *buf, void *cdrarg, int kmflags)
3867 {
3868     mptsas_cmd_t *cmd = buf;
3869     mptsas_t *mpt = cdrarg;
3870     struct scsi_address ap;
3871     uint_t cookiec;
3872     ddi_dma_attr_t arq_dma_attr;
3873     int (*callback)(caddr_t);

3875     callback = (kmflags == KM_SLEEP)? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;

3877     NDBG4(("mptsas_kmem_cache_constructor"));

3879     ap.a_hba_tran = mpt->m_tran;
3880     ap.a_target = 0;
3881     ap.a_lun = 0;

3883     /*
3884     * allocate a dma handle
3885     */
3886     if ((ddi_dma_alloc_handle(mpt->m_dip, &mpt->m_io_dma_attr, callback,
3887         NULL, &cmd->cmd_dmahandle) != DDI_SUCCESS) {
3888         cmd->cmd_dmahandle = NULL;
3889         return (-1);
3890     }

3892     cmd->cmd_arq_buf = scsi_alloc_consistent_buf(&ap, (struct buf *)NULL,
3893         SENSE_LENGTH, B_READ, callback, NULL);
3894     if (cmd->cmd_arq_buf == NULL) {
3895         ddi_dma_free_handle(&cmd->cmd_dmahandle);
3896         cmd->cmd_dmahandle = NULL;
3897         return (-1);
3898     }

3900     /*
3901     * allocate a arq handle
3902     */
3903     arq_dma_attr = mpt->m_msg_dma_attr;
3904     arq_dma_attr.dma_attr_sgllen = 1;
3905     if ((ddi_dma_alloc_handle(mpt->m_dip, &arq_dma_attr, callback,
3906         NULL, &cmd->cmd_arqhandle) != DDI_SUCCESS) {
3907         ddi_dma_free_handle(&cmd->cmd_dmahandle);
3908         scsi_free_consistent_buf(cmd->cmd_arq_buf);
3909         cmd->cmd_dmahandle = NULL;
3910         cmd->cmd_arqhandle = NULL;
3911         return (-1);
3912     }

3914     if (ddi_dma_buf_bind_handle(cmd->cmd_arqhandle,
3915         cmd->cmd_arq_buf, (DDI_DMA_READ | DDI_DMA_CONSISTENT),
3916         callback, NULL, &cmd->cmd_arqcookie, &cookiec) != DDI_SUCCESS) {
3917         ddi_dma_free_handle(&cmd->cmd_dmahandle);
3918         ddi_dma_free_handle(&cmd->cmd_arqhandle);
3919         scsi_free_consistent_buf(cmd->cmd_arq_buf);
3920         cmd->cmd_dmahandle = NULL;

```

```

3921         cmd->cmd_arqhandle = NULL;
3922         cmd->cmd_arq_buf = NULL;
3923         return (-1);
3924     }
3926     return (0);
3927 }

3929 static void
3930 mptsas_kmem_cache_destructor(void *buf, void *cdrarg)
3931 {
3932 #ifndef __lock_lint
3933     _NOTE(ARGUNUSED(cdrarg))
3934 #endif
3935     mptsas_cmd_t *cmd = buf;

3937     NDBG4(("mptsas_kmem_cache_destructor"));

3939     if (cmd->cmd_arqhandle) {
3940         (void) ddi_dma_unbind_handle(cmd->cmd_arqhandle);
3941         ddi_dma_free_handle(&cmd->cmd_arqhandle);
3942         cmd->cmd_arqhandle = NULL;
3943     }
3944     if (cmd->cmd_arq_buf) {
3945         scsi_free_consistent_buf(cmd->cmd_arq_buf);
3946         cmd->cmd_arq_buf = NULL;
3947     }
3948     if (cmd->cmd_dmahandle) {
3949         ddi_dma_free_handle(&cmd->cmd_dmahandle);
3950         cmd->cmd_dmahandle = NULL;
3951     }
3952 }

3954 static int
3955 mptsas_cache_frames_constructor(void *buf, void *cdrarg, int kmflags)
3956 {
3957     mptsas_cache_frames_t *p = buf;
3958     mptsas_t *mpt = cdrarg;
3959     ddi_dma_attr_t frame_dma_attr;
3960     size_t mem_size, alloc_len;
3961     ddi_dma_cookie_t cookie;
3962     uint_t ncookie;
3963     int (*callback)(caddr_t) = (kmflags == KM_SLEEP)
3964         ? DDI_DMA_SLEEP : DDI_DMA_DONTWAIT;

3966     frame_dma_attr = mpt->m_msg_dma_attr;
3967     frame_dma_attr.dma_attr_align = 0x10;
3968     frame_dma_attr.dma_attr_sgllen = 1;

3970     if (ddi_dma_alloc_handle(mpt->m_dip, &frame_dma_attr, callback, NULL,
3971         &p->m_dma_hdl) != DDI_SUCCESS) {
3972         mptsas_log(mpt, CE_WARN, "Unable to allocate dma handle for"
3973             " extra SGL.");
3974         return (DDI_FAILURE);
3975     }

3977     mem_size = (mpt->m_max_request_frames - 1) * mpt->m_req_frame_size;

3979     if (ddi_dma_mem_alloc(p->m_dma_hdl, mem_size, &mpt->m_dev_acc_attr,
3980         DDI_DMA_CONSISTENT, callback, NULL, (caddr_t *)&p->m_frames_addr,
3981         &alloc_len, &p->m_acc_hdl) != DDI_SUCCESS) {
3982         ddi_dma_free_handle(&p->m_dma_hdl);
3983         p->m_dma_hdl = NULL;
3984         mptsas_log(mpt, CE_WARN, "Unable to allocate dma memory for"
3985             " extra SGL.");
3986         return (DDI_FAILURE);

```

```

3987     }

3989     if (ddi_dma_addr_bind_handle(p->m_dma_hdl, NULL, p->m_frames_addr,
3990         alloc_len, DDI_DMA_RDWR | DDI_DMA_CONSISTENT, callback, NULL,
3991         &cookie, &ncookie) != DDI_DMA_MAPPED) {
3992         (void) ddi_dma_mem_free(&p->m_acc_hdl);
3993         ddi_dma_free_handle(&p->m_dma_hdl);
3994         p->m_dma_hdl = NULL;
3995         mptsas_log(mpt, CE_WARN, "Unable to bind DMA resources for"
3996             " extra SGL.");
3997         return (DDI_FAILURE);
3998     }

4000     /*
4001     * Store the SGL memory address. This chip uses this
4002     * address to dma to and from the driver. The second
4003     * address is the address mpt uses to fill in the SGL.
4004     */
4005     p->m_phys_addr = cookie.dmac_address;

4007     return (DDI_SUCCESS);
4008 }

4010 static void
4011 mptsas_cache_frames_destructor(void *buf, void *cdrarg)
4012 {
4013 #ifndef __lock_lint
4014     _NOTE(ARGUNUSED(cdrarg))
4015 #endif
4016     mptsas_cache_frames_t *p = buf;
4017     if (p->m_dma_hdl != NULL) {
4018         (void) ddi_dma_unbind_handle(p->m_dma_hdl);
4019         (void) ddi_dma_mem_free(&p->m_acc_hdl);
4020         ddi_dma_free_handle(&p->m_dma_hdl);
4021         p->m_phys_addr = NULL;
4022         p->m_frames_addr = NULL;
4023         p->m_dma_hdl = NULL;
4024         p->m_acc_hdl = NULL;
4025     }

4027 }

4029 /*
4030 * allocate and deallocate external pkt space (ie. not part of mptsas_cmd)
4031 * for non-standard length cdb, pkt_private, status areas
4032 * if allocation fails, then deallocate all external space and the pkt
4033 */
4034 /* ARGSUSED */
4035 static int
4036 mptsas_pkt_alloc_extern(mptsas_t *mpt, mptsas_cmd_t *cmd,
4037     int cmdlen, int tgtlen, int statuslen, int kf)
4038 {
4039     caddr_t cdbp, scbp, tgt;
4040     int (*callback)(caddr_t) = (kf == KM_SLEEP) ?
4041         DDI_DMA_SLEEP : DDI_DMA_DONTWAIT;
4042     struct scsi_address ap;
4043     size_t senselength;
4044     ddi_dma_attr_t ext_arq_dma_attr;
4045     uint_t cookiec;

4047     NDBG3(("mptsas_pkt_alloc_extern: "
4048         "cmd=0x%p cmdlen=%d tgtlen=%d statuslen=%d kf=%x",
4049         (void *)cmd, cmdlen, tgtlen, statuslen, kf));

4051     tgt = cdbp = scbp = NULL;
4052     cmd->cmd_scblen = statuslen;

```

```

4053     cmd->cmd_privlen      = (uchar_t)tgtlen;
4055     if (cmdlen > sizeof (cmd->cmd_cdb)) {
4056         if ((cdbp = kmem_zalloc((size_t)cmdlen, kf)) == NULL) {
4057             goto fail;
4058         }
4059         cmd->cmd_pkt->pkt_cdbp = (opaque_t)cdbp;
4060         cmd->cmd_flags |= CFLAG_CDBEXTERN;
4061     }
4062     if (tgtlen > PKT_PRIV_LEN) {
4063         if ((tgt = kmem_zalloc((size_t)tgtlen, kf)) == NULL) {
4064             goto fail;
4065         }
4066         cmd->cmd_flags |= CFLAG_PRIVEXTERN;
4067         cmd->cmd_pkt->pkt_private = tgt;
4068     }
4069     if (statuslen > EXTCMD_STATUS_SIZE) {
4070         if ((scbp = kmem_zalloc((size_t)statuslen, kf)) == NULL) {
4071             goto fail;
4072         }
4073         cmd->cmd_flags |= CFLAG_SCBEXTERN;
4074         cmd->cmd_pkt->pkt_scbp = (opaque_t)scbp;
4076
4078         /* allocate sense data buf for DMA */
4079         senselength = statuslen - MPTSAS_GET_ITEM_OFF(
4080             struct scsi_arq_status, sts_sensedata);
4081         cmd->cmd_rqslen = (uchar_t)senselength;
4082
4083         ap.a_hba_tran = mpt->m_tran;
4084         ap.a_target = 0;
4085         ap.a_lun = 0;
4086
4087         cmd->cmd_ext_arq_buf = scsi_alloc_consistent_buf(&ap,
4088             (struct buf *)NULL, senselength, B_READ,
4089             callback, NULL);
4090
4091         if (cmd->cmd_ext_arq_buf == NULL) {
4092             goto fail;
4093         }
4094         /* allocate a extern arq handle and bind the buf
4095          */
4096         ext_arq_dma_attr = mpt->m_msg_dma_attr;
4097         ext_arq_dma_attr.dma_attr_sgllen = 1;
4098         if ((ddi_dma_alloc_handle(mpt->m_dip,
4099             &ext_arq_dma_attr, callback,
4100             NULL, &cmd->cmd_ext_arqhandle)) != DDI_SUCCESS) {
4101             goto fail;
4102         }
4103
4104         if (ddi_dma_buf_bind_handle(cmd->cmd_ext_arqhandle,
4105             cmd->cmd_ext_arq_buf, (DDI_DMA_READ | DDI_DMA_CONSISTENT),
4106             callback, NULL, &cmd->cmd_ext_arqcookie,
4107             &cookiec)
4108             != DDI_SUCCESS) {
4109             goto fail;
4110         }
4111         cmd->cmd_flags |= CFLAG_EXTARQBUFVALID;
4112     }
4113     return (0);
4114 fail:
4115     mptsas_pkt_destroy_extern(mpt, cmd);
4116     return (1);
4117 }

```

```

4119 /*
4120  * deallocate external pkt space and deallocate the pkt
4121  */
4122 static void
4123 mptsas_pkt_destroy_extern(mptsas_t *mpt, mptsas_cmd_t *cmd)
4124 {
4125     NDBG3(("mptsas_pkt_destroy_extern: cmd=0x%p", (void *)cmd));
4127     if (cmd->cmd_flags & CFLAG_FREE) {
4128         mptsas_log(mpt, CE_PANIC,
4129             "mptsas_pkt_destroy_extern: freeing free packet");
4130         _NOTE(NOT_REACHED)
4131         /* NOTREACHED */
4132     }
4133     if (cmd->cmd_flags & CFLAG_CDBEXTERN) {
4134         kmem_free(cmd->cmd_pkt->pkt_cdbp, (size_t)cmd->cmd_cdblen);
4135     }
4136     if (cmd->cmd_flags & CFLAG_SCBEXTERN) {
4137         kmem_free(cmd->cmd_pkt->pkt_scbp, (size_t)cmd->cmd_scblen);
4138         if (cmd->cmd_flags & CFLAG_EXTARQBUFVALID) {
4139             (void) ddi_dma_unbind_handle(cmd->cmd_ext_arqhandle);
4140         }
4141         if (cmd->cmd_ext_arqhandle) {
4142             ddi_dma_free_handle(&cmd->cmd_ext_arqhandle);
4143             cmd->cmd_ext_arqhandle = NULL;
4144         }
4145         if (cmd->cmd_ext_arq_buf)
4146             scsi_free_consistent_buf(cmd->cmd_ext_arq_buf);
4147     }
4148     if (cmd->cmd_flags & CFLAG_PRIVEXTERN) {
4149         kmem_free(cmd->cmd_pkt->pkt_private, (size_t)cmd->cmd_privlen);
4150     }
4151     cmd->cmd_flags = CFLAG_FREE;
4152     kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
4153 }
4155 /*
4156  * tran_sync_pkt(9E) - explicit DMA synchronization
4157  */
4158 /*ARGSUSED*/
4159 static void
4160 mptsas_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
4161 {
4162     mptsas_cmd_t *cmd = PKT2CMD(pkt);
4164     NDBG3(("mptsas_scsi_sync_pkt: target=%d, pkt=0x%p",
4165         ap->a_target, (void *)pkt));
4167     if (cmd->cmd_dmahandle) {
4168         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
4169             (cmd->cmd_flags & CFLAG_DMASEND) ?
4170             DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
4171     }
4172 }
4174 /*
4175  * tran_dmafree(9E) - deallocate DMA resources allocated for command
4176  */
4177 /*ARGSUSED*/
4178 static void
4179 mptsas_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
4180 {
4181     mptsas_cmd_t *cmd = PKT2CMD(pkt);
4182     mptsas_t *mpt = ADDR2MPT(ap);
4184     NDBG3(("mptsas_scsi_dmafree: target=%d pkt=0x%p",

```



```

4185     ap->a_target, (void *)pkt));
4187     if (cmd->cmd_flags & CFLAG_DMAVALID) {
4188         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
4189         cmd->cmd_flags &= ~CFLAG_DMAVALID;
4190     }
4192     if (cmd->cmd_flags & CFLAG_EXTARQBUFVALID) {
4193         (void) ddi_dma_unbind_handle(cmd->cmd_ext_arqhandle);
4194         cmd->cmd_flags &= ~CFLAG_EXTARQBUFVALID;
4195     }
4197     mptsas_free_extra_sgl_frame(mpt, cmd);
4198 }

4200 static void
4201 mptsas_pkt_comp(struct scsi_pkt *pkt, mptsas_cmd_t *cmd)
4202 {
4203     if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
4204         (!(cmd->cmd_flags & CFLAG_DMASEND))) {
4205         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
4206             DDI_DMA_SYNC_FORCPU);
4207     }
4208     (*pkt->pkt_comp)(pkt);
4209 }

4211 static void
4212 mptsas_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
4213     ddi_acc_handle_t acc_hdl, uint_t cookiec, uint32_t end_flags)
4214 {
4215     pMpi2SGESimple64_t     sge;
4216     mpptti_t              *dmap;
4217     uint32_t               flags;

4219     dmap = cmd->cmd_sg;

4221     sge = (pMpi2SGESimple64_t)(&frame->SGL);
4222     while (cookiec--) {
4223         ddi_put32(acc_hdl,
4224             &sge->Address.Low, dmap->addr.address64.Low);
4225         ddi_put32(acc_hdl,
4226             &sge->Address.High, dmap->addr.address64.High);
4227         ddi_put32(acc_hdl, &sge->FlagsLength,
4228             dmap->count);
4229         flags = ddi_get32(acc_hdl, &sge->FlagsLength);
4230         flags |= ((uint32_t)
4231             (MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4232             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4233             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4234             MPI2_SGE_FLAGS_SHIFT);

4236     /*
4237     * If this is the last cookie, we set the flags
4238     * to indicate so
4239     */
4240     if (cookiec == 0) {
4241         flags |= end_flags;
4242     }
4243     if (cmd->cmd_flags & CFLAG_DMASEND) {
4244         flags |= (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4245             MPI2_SGE_FLAGS_SHIFT);
4246     } else {
4247         flags |= (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4248             MPI2_SGE_FLAGS_SHIFT);
4249     }
4250     ddi_put32(acc_hdl, &sge->FlagsLength, flags);

```

```

4251         dmap++;
4252         sge++;
4253     }
4254 }

4256 static void
4257 mptsas_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4258     pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4259 {
4260     pMpi2SGESimple64_t     sge;
4261     pMpi2SGEChain64_t      sgechain;
4262     uint_t                  cookiec;
4263     mpptti_t                *dmap;
4264     uint32_t                 flags;

4266     /*
4267     * Save the number of entries in the DMA
4268     * Scatter/Gather list
4269     */
4270     cookiec = cmd->cmd_cookiec;

4272     /*
4273     * Hereby we start to deal with multiple frames.
4274     * The process is as follows:
4275     * 1. Determine how many frames are needed for SGL element
4276     *    storage; Note that all frames are stored in contiguous
4277     *    memory space and in 64-bit DMA mode each element is
4278     *    3 double-words (12 bytes) long.
4279     * 2. Fill up the main frame. We need to do this separately
4280     *    since it contains the SCSI IO request header and needs
4281     *    dedicated processing. Note that the last 4 double-words
4282     *    of the SCSI IO header is for SGL element storage
4283     *    (MPI2_SGE_IO_UNION).
4284     * 3. Fill the chain element in the main frame, so the DMA
4285     *    engine can use the following frames.
4286     * 4. Enter a loop to fill the remaining frames. Note that the
4287     *    last frame contains no chain element. The remaining
4288     *    frames go into the mpt SGL buffer allocated on the fly,
4289     *    not immediately following the main message frame, as in
4290     *    Gen1.
4291     * Some restrictions:
4292     * 1. For 64-bit DMA, the simple element and chain element
4293     *    are both of 3 double-words (12 bytes) in size, even
4294     *    though all frames are stored in the first 4G of mem
4295     *    range and the higher 32-bits of the address are always 0.
4296     * 2. On some controllers (like the 1064/1068), a frame can
4297     *    hold SGL elements with the last 1 or 2 double-words
4298     *    (4 or 8 bytes) un-used. On these controllers, we should
4299     *    recognize that there's not enough room for another SGL
4300     *    element and move the sge pointer to the next frame.
4301     */
4302     int                i, j, k, l, frames, sgemax;
4303     int                 temp;
4304     uint8_t             chainflags;
4305     uint16_t            chainlength;
4306     mptsas_cache_frames_t *p;

4308     /*
4309     * Sgemax is the number of SGE's that will fit
4310     * each extra frame and frames is total
4311     * number of frames we'll need. 1 sge entry per
4312     * frame is reserved for the chain element thus the -1 below.
4313     */
4314     sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_SGE_SIMPLE64))
4315         - 1);
4316     temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;

```

```

4318  /*
4319  * A little check to see if we need to round up the number
4320  * of frames we need
4321  */
4322  if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
4323      sgemax) > 1) {
4324      frames = (temp + 1);
4325  } else {
4326      frames = temp;
4327  }
4328  dmap = cmd->cmd_sg;
4329  sge = (pMpi2SGESimple64_t)(amp->SGL);

4331  /*
4332  * First fill in the main frame
4333  */
4334  j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
4335  mptsas_sge_mainframe(cmd, frame, acc_hdl, j,
4336      ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4337      MPI2_SGE_FLAGS_SHIFT));
4338  dmap += j;
4339  sge += j;
4340  j++;

4342  /*
4343  * Fill in the chain element in the main frame.
4344  * About calculation on ChainOffset:
4345  * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4346  *   in the end reserved for SGL element storage
4347  *   (MPI2_SGE_IO_UNION); we should count it in our
4348  *   calculation. See its definition in the header file.
4349  * 2. Constant j is the counter of the current SGL element
4350  *   that will be processed, and (j - 1) is the number of
4351  *   SGL elements that have been processed (stored in the
4352  *   main frame).
4353  * 3. ChainOffset value should be in units of double-words (4
4354  *   bytes) so the last value should be divided by 4.
4355  */
4356  ddi_put8(acc_hdl, amp->ChainOffset,
4357      (sizeof(MPI2_SCSI_IO_REQUEST) -
4358      sizeof(MPI2_SGE_IO_UNION) +
4359      (j - 1) * sizeof(MPI2_SGE_SIMPLE64)) >> 2);
4360  sgechain = (pMpi2SGEChain64_t)sge;
4361  chainflags = (MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4362      MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4363      MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4364  ddi_put8(acc_hdl, amp->Flags, chainflags);

4366  /*
4367  * The size of the next frame is the accurate size of space
4368  * (in bytes) used to store the SGL elements. j is the counter
4369  * of SGL elements. (j - 1) is the number of SGL elements that
4370  * have been processed (stored in frames).
4371  */
4372  if (frames >= 2) {
4373      ASSERT(mpt->m_req_frame_size >= sizeof(MPI2_SGE_SIMPLE64));
4374      chainlength = mpt->m_req_frame_size /
4375          sizeof(MPI2_SGE_SIMPLE64) *
4376          sizeof(MPI2_SGE_SIMPLE64);
4377  } else {
4378      chainlength = ((cookiec - (j - 1)) *
4379          sizeof(MPI2_SGE_SIMPLE64));
4380  }

4382  p = cmd->cmd_extra_frames;

```

```

4384  ddi_put16(acc_hdl, amp->Length, chainlength);
4385  ddi_put32(acc_hdl, amp->Address.Low,
4386      p->m_phys_addr);
4387  /* SGL is allocated in the first 4G mem range */
4388  ddi_put32(acc_hdl, amp->Address.High, 0);

4390  /*
4391  * If there are more than 2 frames left we have to
4392  * fill in the next chain offset to the location of
4393  * the chain element in the next frame.
4394  * sgemax is the number of simple elements in an extra
4395  * frame. Note that the value NextChainOffset should be
4396  * in double-words (4 bytes).
4397  */
4398  if (frames >= 2) {
4399      ddi_put8(acc_hdl, amp->NextChainOffset,
4400          (sgemax * sizeof(MPI2_SGE_SIMPLE64)) >> 2);
4401  } else {
4402      ddi_put8(acc_hdl, amp->NextChainOffset, 0);
4403  }

4405  /*
4406  * Jump to next frame;
4407  * Starting here, chain buffers go into the per command SGL.
4408  * This buffer is allocated when chain buffers are needed.
4409  */
4410  sge = (pMpi2SGESimple64_t)p->m_frames_addr;
4411  i = cookiec;

4413  /*
4414  * Start filling in frames with SGE's. If we
4415  * reach the end of frame and still have SGE's
4416  * to fill we need to add a chain element and
4417  * use another frame. j will be our counter
4418  * for what cookie we are at and i will be
4419  * the total cookiec. k is the current frame
4420  */
4421  for (k = 1; k <= frames; k++) {
4422      for (l = 1; l <= (sgemax + 1) && (j <= i); j++, l++) {

4424          /*
4425          * If we have reached the end of frame
4426          * and we have more SGE's to fill in
4427          * we have to fill the final entry
4428          * with a chain element and then
4429          * continue to the next frame
4430          */
4431          if ((l == (sgemax + 1)) && (k != frames)) {
4432              sgechain = (pMpi2SGEChain64_t)sge;
4433              j--;
4434              chainflags = (
4435                  MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4436                  MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4437                  MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4438              ddi_put8(p->m_acc_hdl,
4439                  amp->Flags, chainflags);
4440          /*
4441          * k is the frame counter and (k + 1)
4442          * is the number of the next frame.
4443          * Note that frames are in contiguous
4444          * memory space.
4445          */
4446              ddi_put32(p->m_acc_hdl,
4447                  amp->Address.Low,
4448                  (p->m_phys_addr +

```

```

4449         (mpt->m_req_frame_size * k));
4450         ddi_put32(p->m_acc_hdl,
4451             &sgechain->Address.High, 0);
4452
4453         /*
4454          * If there are more than 2 frames left
4455          * we have to next chain offset to
4456          * the location of the chain element
4457          * in the next frame and fill in the
4458          * length of the next chain
4459          */
4460         if ((frames - k) >= 2) {
4461             ddi_put8(p->m_acc_hdl,
4462                 &sgechain->NextChainOffset,
4463                 (sgemax *
4464                 sizeof (MPI2_SGE_SIMPLE64))
4465                 >> 2);
4466             ddi_put16(p->m_acc_hdl,
4467                 &sgechain->Length,
4468                 mpt->m_req_frame_size /
4469                 sizeof (MPI2_SGE_SIMPLE64) *
4470                 sizeof (MPI2_SGE_SIMPLE64));
4471         } else {
4472             /*
4473              * This is the last frame. Set
4474              * the NextChainOffset to 0 and
4475              * Length is the total size of
4476              * all remaining simple elements
4477              */
4478             ddi_put8(p->m_acc_hdl,
4479                 &sgechain->NextChainOffset,
4480                 0);
4481             ddi_put16(p->m_acc_hdl,
4482                 &sgechain->Length,
4483                 (cookiec - j) *
4484                 sizeof (MPI2_SGE_SIMPLE64));
4485         }
4486
4487         /* Jump to the next frame */
4488         sge = (pMpi2SGESimple64_t)
4489             ((char *)p->m_frames_addr +
4490              (int)mpt->m_req_frame_size * k);
4491
4492         continue;
4493     }
4494
4495     ddi_put32(p->m_acc_hdl,
4496         &sge->Address.Low,
4497         dmap->addr.address64.Low);
4498     ddi_put32(p->m_acc_hdl,
4499         &sge->Address.High,
4500         dmap->addr.address64.High);
4501     ddi_put32(p->m_acc_hdl,
4502         &sge->FlagsLength, dmap->count);
4503     flags = ddi_get32(p->m_acc_hdl,
4504         &sge->FlagsLength);
4505     flags |= ((uint32_t)(
4506         MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4507         MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4508         MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4509         MPI2_SGE_FLAGS_SHIFT);
4510
4511     /*
4512      * If we are at the end of the frame and
4513      * there is another frame to fill in
4514      * we set the last simple element as last

```

```

4515         * element
4516         */
4517         if ((l == sgemax) && (k != frames)) {
4518             flags |= ((uint32_t)
4519                 (MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4520                 MPI2_SGE_FLAGS_SHIFT);
4521         }
4522
4523         /*
4524          * If this is the final cookie we
4525          * indicate it by setting the flags
4526          */
4527         if (j == i) {
4528             flags |= ((uint32_t)
4529                 (MPI2_SGE_FLAGS_LAST_ELEMENT |
4530                 MPI2_SGE_FLAGS_END_OF_BUFFER |
4531                 MPI2_SGE_FLAGS_END_OF_LIST) <<
4532                 MPI2_SGE_FLAGS_SHIFT);
4533         }
4534         if (cmd->cmd_flags & CFLAG_DMASEND) {
4535             flags |=
4536                 (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4537                 MPI2_SGE_FLAGS_SHIFT);
4538         } else {
4539             flags |=
4540                 (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4541                 MPI2_SGE_FLAGS_SHIFT);
4542         }
4543         ddi_put32(p->m_acc_hdl,
4544             &sge->FlagsLength, flags);
4545         dmap++;
4546         sge++;
4547     }
4548 }
4549
4550     /*
4551      * Sync DMA with the chain buffers that were just created
4552      */
4553     (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4554 }
4555
4556 static void
4557 mptsas_ieee_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
4558     ddi_acc_handle_t acc_hdl, uint_t cookiec, uint8_t end_flag)
4559 {
4560     pMpi2IeeeSgeSimple64_t ieesge;
4561     mptti_t *dmap;
4562     uint8_t flags;
4563
4564     dmap = cmd->cmd_sg;
4565
4566     NDBG1(("mptsas_ieee_sge_mainframe: cookiec=%d, %s", cookiec,
4567         cmd->cmd_flags & CFLAG_DMASEND?"Out":"In"));
4568
4569     ieesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);
4570     while (cookiec--) {
4571         ddi_put32(acc_hdl,
4572             &ieeesge->Address.Low, dmap->addr.address64.Low);
4573         ddi_put32(acc_hdl,
4574             &ieeesge->Address.High, dmap->addr.address64.High);
4575         ddi_put32(acc_hdl, &ieeesge->Length,
4576             dmap->count);
4577         NDBG1(("mptsas_ieee_sge_mainframe: len=%d", dmap->count));
4578         flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4579             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);

```

```

4581      /*
4582      * If this is the last cookie, we set the flags
4583      * to indicate so
4584      */
4585      if (cookiec == 0) {
4586          flags |= end_flag;
4587      }

4589      ddi_put8(acc_hdl, &ieeesge->Flags, flags);
4590      dmap++;
4591      ieeesge++;
4592  }
4593 }

4595 static void
4596 mptsas_ieee_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4597     pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4598 {
4599     pMpi2IeeeSgeSimple64_t ieeesge;
4600     pMpi25IeeeSgeChain64_t ieeesgechain;
4601     uint_t cookiec;
4602     mpptti_t *dmap;
4603     uint8_t flags;

4605     /*
4606     * Save the number of entries in the DMA
4607     * Scatter/Gather list
4608     */
4609     cookiec = cmd->cmd_cookiec;

4611     NDBG1(("mptsas_ieee_sge_chain: cookiec=%d", cookiec));

4613     /*
4614     * Hereby we start to deal with multiple frames.
4615     * The process is as follows:
4616     * 1. Determine how many frames are needed for SGL element
4617     *    storage; Note that all frames are stored in contiguous
4618     *    memory space and in 64-bit DMA mode each element is
4619     *    4 double-words (16 bytes) long.
4620     * 2. Fill up the main frame. We need to do this separately
4621     *    since it contains the SCSI IO request header and needs
4622     *    dedicated processing. Note that the last 4 double-words
4623     *    of the SCSI IO header is for SGL element storage
4624     *    (MPI2_SGE_IO_UNION).
4625     * 3. Fill the chain element in the main frame, so the DMA
4626     *    engine can use the following frames.
4627     * 4. Enter a loop to fill the remaining frames. Note that the
4628     *    last frame contains no chain element. The remaining
4629     *    frames go into the mpt SGL buffer allocated on the fly,
4630     *    not immediately following the main message frame, as in
4631     *    Gen1.
4632     * Restrictions:
4633     * For 64-bit DMA, the simple element and chain element
4634     * are both of 4 double-words (16 bytes) in size, even
4635     * though all frames are stored in the first 4G of mem
4636     * range and the higher 32-bits of the address are always 0.
4637     */
4638     int i, j, k, l, frames, sgemax;
4639     int temp;
4640     uint8_t chainflags;
4641     uint32_t chainlength;
4642     mptsas_cache_frames_t *p;

4644     /*
4645     * Sgemax is the number of SGE's that will fit
4646     * each extra frame and frames is total

```

```

4647     * number of frames we'll need. 1 sge entry per
4648     * frame is reserved for the chain element thus the -1 below.
4649     */
4650     sgemax = ((mpt->m_req_frame_size / sizeof(MPI2_IEEE_SGE_SIMPLE64))
4651         - 1);
4652     temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;

4654     /*
4655     * A little check to see if we need to round up the number
4656     * of frames we need
4657     */
4658     if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
4659         sgemax) > 1) {
4660         frames = (temp + 1);
4661     } else {
4662         frames = temp;
4663     }
4664     NDBG1(("mptsas_ieee_sge_chain: temp=%d, frames=%d", temp, frames));
4665     dmap = cmd->cmd_sg;
4666     ieeesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);

4668     /*
4669     * First fill in the main frame
4670     */
4671     j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
4672     mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl, j, 0);
4673     dmap += j;
4674     ieeesge += j;
4675     j++;

4677     /*
4678     * Fill in the chain element in the main frame.
4679     * About calculation on ChainOffset:
4680     * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4681     *    in the end reserved for SGL element storage
4682     *    (MPI2_SGE_IO_UNION); we should count it in our
4683     *    calculation. See its definition in the header file.
4684     * 2. Constant j is the counter of the current SGL element
4685     *    that will be processed, and (j - 1) is the number of
4686     *    SGL elements that have been processed (stored in the
4687     *    main frame).
4688     * 3. ChainOffset value should be in units of quad-words (16
4689     *    bytes) so the last value should be divided by 16.
4690     */
4691     ddi_put8(acc_hdl, &frame->ChainOffset,
4692         (sizeof(MPI2_SCSI_IO_REQUEST) -
4693         sizeof(MPI2_SGE_IO_UNION) +
4694         (j - 1) * sizeof(MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4695     ieeesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
4696     chainflags = (MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4697         MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4698     ddi_put8(acc_hdl, &ieeesgechain->Flags, chainflags);

4700     /*
4701     * The size of the next frame is the accurate size of space
4702     * (in bytes) used to store the SGL elements. j is the counter
4703     * of SGL elements. (j - 1) is the number of SGL elements that
4704     * have been processed (stored in frames).
4705     */
4706     if (frames >= 2) {
4707         ASSERT(mpt->m_req_frame_size >=
4708             sizeof(MPI2_IEEE_SGE_SIMPLE64));
4709         chainlength = mpt->m_req_frame_size /
4710             sizeof(MPI2_IEEE_SGE_SIMPLE64) *
4711             sizeof(MPI2_IEEE_SGE_SIMPLE64);
4712     } else {

```

```

4713         chainlength = ((cookiec - (j - 1)) *
4714             sizeof (MPI2_IEEE_SGE_SIMPLE64));
4715     }
4717     p = cmd->cmd_extra_frames;
4719     ddi_put32(acc_hdl, &ieeesgechain->Length, chainlength);
4720     ddi_put32(acc_hdl, &ieeesgechain->Address.Low,
4721         p->m_phys_addr);
4722     /* SGL is allocated in the first 4G mem range */
4723     ddi_put32(acc_hdl, &ieeesgechain->Address.High, 0);
4725     /*
4726      * If there are more than 2 frames left we have to
4727      * fill in the next chain offset to the location of
4728      * the chain element in the next frame.
4729      * sgemax is the number of simple elements in an extra
4730      * frame. Note that the value NextChainOffset should be
4731      * in double-words (4 bytes).
4732      */
4733     if (frames >= 2) {
4734         ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset,
4735             (sgemax * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4736     } else {
4737         ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset, 0);
4738     }
4740     /*
4741      * Jump to next frame;
4742      * Starting here, chain buffers go into the per command SGL.
4743      * This buffer is allocated when chain buffers are needed.
4744      */
4745     ieesge = (pMpi2IeeeSgeSimple64_t)p->m_frames_addr;
4746     i = cookiec;
4748     /*
4749      * Start filling in frames with SGE's. If we
4750      * reach the end of frame and still have SGE's
4751      * to fill we need to add a chain element and
4752      * use another frame. j will be our counter
4753      * for what cookie we are at and i will be
4754      * the total cookiec. k is the current frame
4755      */
4756     for (k = 1; k <= frames; k++) {
4757         for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {
4759             /*
4760              * If we have reached the end of frame
4761              * and we have more SGE's to fill in
4762              * we have to fill the final entry
4763              * with a chain element and then
4764              * continue to the next frame
4765              */
4766             if ((l == (sgemax + 1)) && (k != frames)) {
4767                 ieesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
4768                 j--;
4769                 chainflags =
4770                     MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4771                     MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR;
4772                 ddi_put8(p->m_acc_hdl,
4773                     &ieeesgechain->Flags, chainflags);
4774                 /*
4775                  * k is the frame counter and (k + 1)
4776                  * is the number of the next frame.
4777                  * Note that frames are in contiguous
4778                  * memory space.

```

```

4779         */
4780         ddi_put32(p->m_acc_hdl,
4781             &ieeesgechain->Address.Low,
4782             (p->m_phys_addr +
4783                 (mpt->m_req_frame_size * k)));
4784         ddi_put32(p->m_acc_hdl,
4785             &ieeesgechain->Address.High, 0);
4787     /*
4788      * If there are more than 2 frames left
4789      * we have to next chain offset to
4790      * the location of the chain element
4791      * in the next frame and fill in the
4792      * length of the next chain
4793      */
4794     if ((frames - k) >= 2) {
4795         ddi_put8(p->m_acc_hdl,
4796             &ieeesgechain->NextChainOffset,
4797             (sgemax *
4798                 sizeof (MPI2_IEEE_SGE_SIMPLE64))
4799             >> 4);
4800         ASSERT(mpt->m_req_frame_size >=
4801             sizeof (MPI2_IEEE_SGE_SIMPLE64));
4802         ddi_put32(p->m_acc_hdl,
4803             &ieeesgechain->Length,
4804             mpt->m_req_frame_size /
4805             sizeof (MPI2_IEEE_SGE_SIMPLE64) *
4806             sizeof (MPI2_IEEE_SGE_SIMPLE64));
4807     } else {
4808         /*
4809          * This is the last frame. Set
4810          * the NextChainOffset to 0 and
4811          * Length is the total size of
4812          * all remaining simple elements
4813          */
4814         ddi_put8(p->m_acc_hdl,
4815             &ieeesgechain->NextChainOffset,
4816             0);
4817         ddi_put32(p->m_acc_hdl,
4818             &ieeesgechain->Length,
4819             (cookiec - j) *
4820             sizeof (MPI2_IEEE_SGE_SIMPLE64));
4821     }
4823     /* Jump to the next frame */
4824     ieesge = (pMpi2IeeeSgeSimple64_t)
4825         ((char *)p->m_frames_addr +
4826             (int)mpt->m_req_frame_size * k);
4828     continue;
4829 }
4831     ddi_put32(p->m_acc_hdl,
4832         &ieeesge->Address.Low,
4833         dmap->addr.address64.Low);
4834     ddi_put32(p->m_acc_hdl,
4835         &ieeesge->Address.High,
4836         dmap->addr.address64.High);
4837     ddi_put32(p->m_acc_hdl,
4838         &ieeesge->Length, dmap->count);
4839     flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4840         MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4842     /*
4843      * If we are at the end of the frame and
4844      * there is another frame to fill in

```

```

4845     * do we need to do anything?
4846     * if ((l == sgemax) && (k != frames)) {
4847     * }
4848     */
4850     /*
4851     * If this is the final cookie set end of list.
4852     */
4853     if (j == i) {
4854         flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
4855     }
4857     ddi_put8(p->m_acc_hdl, &ieeesge->Flags, flags);
4858     dmap++;
4859     ieeesge++;
4860 }
4861 }
4863 /*
4864 * Sync DMA with the chain buffers that were just created
4865 */
4866 (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4867 }
4869 static void
4870 mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd, uint32_t *control,
4871                pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4872 {
4873     ASSERT(cmd->cmd_flags & CFLAG_DMAVALID);
4875     NDBG1(("mptsas_sge_setup: cookiec=%d", cmd->cmd_cookiec));
4877     /*
4878     * Set read/write bit in control.
4879     */
4880     if (cmd->cmd_flags & CFLAG_DMASEND) {
4881         *control |= MPI2_SCSIIO_CONTROL_WRITE;
4882     } else {
4883         *control |= MPI2_SCSIIO_CONTROL_READ;
4884     }
4886     ddi_put32(acc_hdl, &frame->DataLength, cmd->cmd_dmacount);
4888     /*
4889     * We have 4 cases here. First where we can fit all the
4890     * SG elements into the main frame, and the case
4891     * where we can't. The SG element is also different when using
4892     * MPI2.5 interface.
4893     * If we have more cookies than we can attach to a frame
4894     * we will need to use a chain element to point
4895     * a location of memory where the rest of the S/G
4896     * elements reside.
4897     */
4898     if (cmd->cmd_cookiec <= MPTSAS_MAX_FRAME_SGES64(mpt)) {
4899         if (mpt->m_MPI25) {
4900             mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl,
4901                                     cmd->cmd_cookiec,
4902                                     MPI25_IEEE_SGE_FLAGS_END_OF_LIST);
4903         } else {
4904             mptsas_sge_mainframe(cmd, frame, acc_hdl,
4905                                 cmd->cmd_cookiec,
4906                                 ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT
4907 | MPI2_SGE_FLAGS_END_OF_BUFFER
4908 | MPI2_SGE_FLAGS_END_OF_LIST) <<
4909                                 MPI2_SGE_FLAGS_SHIFT));
4910         }

```

```

4911     } else {
4912         if (mpt->m_MPI25) {
4913             mptsas_ieee_sge_chain(mpt, cmd, frame, acc_hdl);
4914         } else {
4915             mptsas_sge_chain(mpt, cmd, frame, acc_hdl);
4916         }
4917     }
4918 }
4920 /*
4921 * Interrupt handling
4922 * Utility routine. Poll for status of a command sent to HBA
4923 * without interrupts (a FLAG_NOINTR command).
4924 */
4925 int
4926 mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
4927 {
4928     int        rval = TRUE;
4930     NDBG5(("mptsas_poll: cmd=0x%p", (void *)poll_cmd));
4932     if ((poll_cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
4933         mptsas_restart_hba(mpt);
4934     }
4936     /*
4937     * Wait, using drv_usecwait(), long enough for the command to
4938     * reasonably return from the target if the target isn't
4939     * "dead". A polled command may well be sent from scsi_poll, and
4940     * there are retries built in to scsi_poll if the transport
4941     * accepted the packet (TRAN_ACCEPT). scsi_poll waits 1 second
4942     * and retries the transport up to scsi_poll_busycnt times
4943     * (currently 60) if
4944     * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
4945     * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
4946     *
4947     * limit the waiting to avoid a hang in the event that the
4948     * cmd never gets started but we are still receiving interrupts
4949     */
4950     while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
4951         if (mptsas_wait_intr(mpt, polltime) == FALSE) {
4952             NDBG5(("mptsas_poll: command incomplete"));
4953             rval = FALSE;
4954             break;
4955         }
4956     }
4958     if (rval == FALSE) {
4960         /*
4961         * this isn't supposed to happen, the hba must be wedged
4962         * Mark this cmd as a timeout.
4963         */
4964         mptsas_set_pkt_reason(mpt, poll_cmd, CMD_TIMEOUT,
4965                               (STAT_TIMEOUT|STAT_ABORTED));
4967         if (poll_cmd->cmd_queued == FALSE) {
4969             NDBG5(("mptsas_poll: not on waitq"));
4971             poll_cmd->cmd_pkt->pkt_state |=
4972                 (STATE_GOT_BUS|STATE_GOT_TARGET|STATE_SENT_CMD);
4973         } else {
4975             /* find and remove it from the waitq */
4976             NDBG5(("mptsas_poll: delete from waitq"));

```

```

4977         mptsas_waitq_delete(mpt, poll_cmd);
4978     }
4980 }
4981 mptsas_fma_check(mpt, poll_cmd);
4982 NDBG5(("mptsas_poll: done"));
4983 return (rval);
4984 }

4986 /*
4987  * Used for polling cmds and TM function
4988  */
4989 static int
4990 mptsas_wait_intr(mptsas_t *mpt, int polltime)
4991 {
4992     int cnt;
4993     pMpi2ReplyDescriptorsUnion_t reply_desc_union;
4994     uint32_t int_mask;

4996     NDBG5(("mptsas_wait_intr"));

4998     mpt->m_polled_intr = 1;

5000     /*
5001      * Get the current interrupt mask and disable interrupts.  When
5002      * re-enabling ints, set mask to saved value.
5003      */
5004     int_mask = ddi_get32(mpt->m_datap, &mpt->m_reg->HostInterruptMask);
5005     MPTSAS_DISABLE_INTR(mpt);

5007     /*
5008      * Keep polling for at least (polltime * 1000) seconds
5009      */
5010     for (cnt = 0; cnt < polltime; cnt++) {
5011         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5012             DDI_DMA_SYNC_FORCPU);

5014         reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
5015             MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

5017         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5018             &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5019             ddi_get32(mpt->m_acc_post_queue_hdl,
5020             &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5021             drv_usecwait(1000);
5022             continue;
5023         }

5025         /*
5026          * The reply is valid, process it according to its
5027          * type.
5028          */
5029         mptsas_process_intr(mpt, reply_desc_union);

5031         if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5032             mpt->m_post_index = 0;
5033         }

5035         /*
5036          * Update the global reply index
5037          */
5038         ddi_put32(mpt->m_datap,
5039             &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
5040         mpt->m_polled_intr = 0;

5042         /*

```

```

5043         * Re-enable interrupts and quit.
5044         */
5045         ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask,
5046             int_mask);
5047         return (TRUE);

5049     }

5051     /*
5052      * Clear polling flag, re-enable interrupts and quit.
5053      */
5054     mpt->m_polled_intr = 0;
5055     ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, int_mask);
5056     return (FALSE);
5057 }

5059 static void
5060 mptsas_handle_scsi_io_success(mptsas_t *mpt,
5061     pMpi2ReplyDescriptorsUnion_t reply_desc)
5062 {
5063     pMpi2SCSIIOSuccessReplyDescriptor_t scsi_io_success;
5064     uint16_t SMID;
5065     mptsas_slots_t *slots = mpt->m_active;
5066     mptsas_cmd_t *cmd = NULL;
5067     struct scsi_pkt *pkt;

5069     ASSERT(mutex_owned(&mpt->m_mutex));

5071     scsi_io_success = (pMpi2SCSIIOSuccessReplyDescriptor_t)reply_desc;
5072     SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &scsi_io_success->SMID);

5074     /*
5075      * This is a success reply so just complete the IO.  First, do a sanity
5076      * check on the SMID.  The final slot is used for TM requests, which
5077      * would not come into this reply handler.
5078      */
5079     if ((SMID == 0) || (SMID > slots->m_n_normal)) {
5080         mptsas_log(mpt, CE_WARN, "?Received invalid SMID of %d\n",
5081             SMID);
5082         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5083         return;
5084     }

5086     cmd = slots->m_slot[SMID];

5088     /*
5089      * print warning and return if the slot is empty
5090      */
5091     if (cmd == NULL) {
5092         mptsas_log(mpt, CE_WARN, "?NULL command for successful SCSI IO "
5093             "in slot %d", SMID);
5094         return;
5095     }

5097     pkt = CMD2PKT(cmd);
5098     pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET | STATE_SENT_CMD |
5099         STATE_GOT_STATUS);
5100     if (cmd->cmd_flags & CFLAG_DMAVALID) {
5101         pkt->pkt_state |= STATE_XFERRED_DATA;
5102     }
5103     pkt->pkt_resid = 0;

5105     if (cmd->cmd_flags & CFLAG_PASSTHRU) {
5106         cmd->cmd_flags |= CFLAG_FINISHED;
5107         cv_broadcast(&mpt->m_passthru_cv);
5108         return;

```

```

5109     } else {
5110         mptsas_remove_cmd(mpt, cmd);
5111     }

5113     if (cmd->cmd_flags & CFLAG_RETRY) {
5114         /*
5115          * The target returned QFULL or busy, do not add this
5116          * pkt to the doneq since the hba will retry
5117          * this cmd.
5118          *
5119          * The pkt has already been resubmitted in
5120          * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
5121          * Remove this cmd_flag here.
5122          */
5123         cmd->cmd_flags &= ~CFLAG_RETRY;
5124     } else {
5125         mptsas_doneq_add(mpt, cmd);
5126     }
5127 }

5129 static void
5130 mptsas_handle_address_reply(mptsas_t *mpt,
5131     pMpi2ReplyDescriptorsUnion_t reply_desc)
5132 {
5133     pMpi2AddressReplyDescriptor_t    address_reply;
5134     pMPI2DefaultReply_t              reply;
5135     mptsas_fw_diagnostic_buffer_t    *pBuffer;
5136     uint32_t                          reply_addr;
5137     uint16_t                          SMID, iocstatus;
5138     mptsas_slots_t                   *slots = mpt->m_active;
5139     mptsas_cmd_t                     *cmd = NULL;
5140     uint8_t                          function, buffer_type;
5141     m_replyh_arg_t                   *args;
5142     int                                reply_frame_no;

5144     ASSERT(mutex_owned(&mpt->m_mutex));

5146     address_reply = (pMpi2AddressReplyDescriptor_t)reply_desc;
5147     reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
5148         &address_reply->ReplyFrameAddress);
5149     SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

5151     /*
5152      * If reply frame is not in the proper range we should ignore this
5153      * message and exit the interrupt handler.
5154      */
5155     if ((reply_addr < mpt->m_reply_frame_dma_addr) ||
5156         (reply_addr >= (mpt->m_reply_frame_dma_addr +
5157             (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
5158         ((reply_addr - mpt->m_reply_frame_dma_addr) %
5159             mpt->m_reply_frame_size != 0)) {
5160         mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
5161             "address 0x%x\n", reply_addr);
5162         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5163         return;
5164     }

5166     (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
5167         DDI_DMA_SYNC_FORCPU);
5168     reply = (pMPI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
5169         mpt->m_reply_frame_dma_addr));
5170     function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

5172     NDBG31(("mptsas_handle_address_reply: function 0x%x, reply_addr=0x%x",
5173         function, reply_addr));

```

```

5175     /*
5176      * don't get slot information and command for events since these values
5177      * don't exist
5178      */
5179     if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
5180         (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
5181         /*
5182          * This could be a TM reply, which use the last allocated SMID,
5183          * so allow for that.
5184          */
5185         if ((SMID == 0) || (SMID > (slots->m_n_normal + 1))) {
5186             mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
5187                 "%d\n", SMID);
5188             ddi_fm_service_impact(mpt->m_dip,
5189                 DDI_SERVICE_UNAFFECTED);
5190             return;
5191         }

5193         cmd = slots->m_slot[SMID];

5195         /*
5196          * print warning and return if the slot is empty
5197          */
5198         if (cmd == NULL) {
5199             mptsas_log(mpt, CE_WARN, "?NULL command for address "
5200                 "reply in slot %d", SMID);
5201             return;
5202         }
5203         if ((cmd->cmd_flags &
5204             (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG))) {
5205             cmd->cmd_rfm = reply_addr;
5206             cmd->cmd_flags |= CFLAG_FINISHED;
5207             cv_broadcast(&mpt->m_passthru_cv);
5208             cv_broadcast(&mpt->m_config_cv);
5209             cv_broadcast(&mpt->m_fw_diag_cv);
5210             return;
5211         } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
5212             mptsas_remove_cmd(mpt, cmd);
5213         }
5214         NDBG31(("\\t\\tmptsas_process_intr: slot=%d", SMID));
5215     }
5216     /*
5217      * Depending on the function, we need to handle
5218      * the reply frame (and cmd) differently.
5219      */
5220     switch (function) {
5221     case MPI2_FUNCTION_SCSI_IO_REQUEST:
5222         mptsas_check_scsi_io_error(mpt, (pMpi2SCSIIOReply_t)reply, cmd);
5223         break;
5224     case MPI2_FUNCTION_SCSI_TASK_MGMT:
5225         cmd->cmd_rfm = reply_addr;
5226         mptsas_check_task_mgt(mpt, (pMpi2SCSIManagementReply_t)reply,
5227             cmd);
5228         break;
5229     case MPI2_FUNCTION_FW_DOWNLOAD:
5230         cmd->cmd_flags |= CFLAG_FINISHED;
5231         cv_signal(&mpt->m_fw_cv);
5232         break;
5233     case MPI2_FUNCTION_EVENT_NOTIFICATION:
5234         reply_frame_no = (reply_addr - mpt->m_reply_frame_dma_addr) /
5235             mpt->m_reply_frame_size;
5236         args = &mpt->m_replyh_args[reply_frame_no];
5237         args->mpt = (void *)mpt;
5238         args->rfm = reply_addr;

5240         /*

```



```

5241     * Record the event if its type is enabled in
5242     * this mpt instance by iocctl.
5243     */
5244     mptsas_record_event(args);

5246     /*
5247     * Handle time critical events
5248     * NOT_RESPONDING/ADDED only now
5249     */
5250     if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {
5251         /*
5252         * Would not return main process,
5253         * just let taskq resolve ack action
5254         * and ack would be sent in taskq thread
5255         */
5256         NDBG20(("send mptsas_handle_event_sync success"));
5257     }

5259     if (mpt->m_in_reset) {
5260         NDBG20(("dropping event received during reset"));
5261         return;
5262     }

5264     if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
5265         (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
5266         mptsas_log(mpt, CE_WARN, "No memory available"
5267             "for dispatch taskq");
5268         /*
5269         * Return the reply frame to the free queue.
5270         */
5271         ddi_put32(mpt->m_acc_free_queue_hdl,
5272             &((uint32_t *))(void *)
5273             mpt->m_free_queue[mpt->m_free_index], reply_addr);
5274         (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
5275             DDI_DMA_SYNC_FORDEV);
5276         if (++mpt->m_free_index == mpt->m_free_queue_depth) {
5277             mpt->m_free_index = 0;
5278         }

5280         ddi_put32(mpt->m_datap,
5281             &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
5282     }
5283     return;
5284     case MPI2_FUNCTION_DIAG_BUFFER_POST:
5285         /*
5286         * If SMID is 0, this implies that the reply is due to a
5287         * release function with a status that the buffer has been
5288         * released. Set the buffer flags accordingly.
5289         */
5290         if (SMID == 0) {
5291             iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
5292                 &reply->IOCSTATUS);
5293             buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
5294                 &((pMpi2DiagBufferPostReply_t)reply)->BufferType);
5295             if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
5296                 pBuffer =
5297                     &mpt->m_fw_diag_buffer_list[buffer_type];
5298                 pBuffer->valid_data = TRUE;
5299                 pBuffer->owned_by_firmware = FALSE;
5300                 pBuffer->immediate = FALSE;
5301             }
5302         } else {
5303             /*
5304             * Normal handling of diag post reply with SMID.
5305             */
5306             cmd = slots->m_slot[SMID];

```

```

5308         /*
5309         * print warning and return if the slot is empty
5310         */
5311         if (cmd == NULL) {
5312             mptsas_log(mpt, CE_WARN, "?NULL command for "
5313                 "address reply in slot %d", SMID);
5314             return;
5315         }
5316         cmd->cmd_rfm = reply_addr;
5317         cmd->cmd_flags |= CFLAG_FINISHED;
5318         cv_broadcast(&mpt->m_fw_diag_cv);
5319     }
5320     return;
5321     default:
5322         mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
5323         break;
5324     }

5326     /*
5327     * Return the reply frame to the free queue.
5328     */
5329     ddi_put32(mpt->m_acc_free_queue_hdl,
5330         &((uint32_t *))(void *)mpt->m_free_queue[mpt->m_free_index],
5331         reply_addr);
5332     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
5333         DDI_DMA_SYNC_FORDEV);
5334     if (++mpt->m_free_index == mpt->m_free_queue_depth) {
5335         mpt->m_free_index = 0;
5336     }
5337     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
5338         mpt->m_free_index);

5340     if (cmd->cmd_flags & CFLAG_FW_CMD)
5341         return;

5343     if (cmd->cmd_flags & CFLAG_RETRY) {
5344         /*
5345         * The target returned QFULL or busy, do not add this
5346         * pkt to the doneq since the hba will retry
5347         * this cmd.
5348         *
5349         * The pkt has already been resubmitted in
5350         * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
5351         * Remove this cmd_flag here.
5352         */
5353         cmd->cmd_flags &= ~CFLAG_RETRY;
5354     } else {
5355         mptsas_doneq_add(mpt, cmd);
5356     }
5357 }

5359 static void
5360 mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
5361     mptsas_cmd_t *cmd)
5362 {
5363     uint8_t         scsi_status, scsi_state;
5364     uint16_t        ioc_status;
5365     uint32_t        xferred, sensecount, respondedata, loginfo = 0;
5366     struct scsi_pkt *pkt;
5367     struct scsi_arq_status *arqstat;
5368     struct buf      *bp;
5369     mptsas_target_t *tgt = cmd->cmd_tgt_addr;
5370     uint8_t         *sensedata = NULL;
5371     uint64_t        sas_wwn;
5372     uint8_t         phy;

```



```

5505         "resource for handle SAS dynamic"
5506         "reconfigure.\n");
5507         break;
5508     }
5509     topo_node->mpt = mpt;
5510     topo_node->event = MPTSAS_DR_EVENT_RECONFIG_TARGET;
5511     topo_node->un.phymask = ptgt->m_addr.mta_phymask;
5512     topo_node->devhdl = ptgt->m_devhdl;
5513     topo_node->object = (void *)ptgt;
5514     topo_node->flags = MPTSAS_TOPO_FLAG_LUN_ASSOCIATED;

5516     if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
5517         mptsas_handle_dr,
5518         (void *)topo_node,
5519         DDI_NOSLEEP)) != DDI_SUCCESS) {
5520         kmem_free(topo_node,
5521             sizeof(mptsas_topo_change_list_t));
5522         mptsas_log(mpt, CE_NOTE, "mptsas start taskq"
5523             "for handle SAS dynamic reconfigure"
5524             "failed. \n");
5525     }
5526 }
5527 break;
5528 case MPI2_SCSI_STATUS_GOOD:
5529     switch (ioc_status & MPI2_IOCSTATUS_MASK) {
5530     case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
5531         pkt->pkt_reason = CMD_DEV_GONE;
5532         pkt->pkt_state |= STATE_GOT_BUS;
5533         if (ptgt->m_reset_delay == 0) {
5534             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5535         }
5536         NDBG31(("lost disk for target%d, command:%x",
5537             Tgt(cmd), pkt->pkt_cdbp[0]));
5538         break;
5539     case MPI2_IOCSTATUS_SCSI_DATA_OVERRUN:
5540         NDBG31(("data overrun: xferred=%d", xferred));
5541         NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5542         pkt->pkt_reason = CMD_DATA_OVR;
5543         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5544             | STATE_SENT_CMD | STATE_GOT_STATUS
5545             | STATE_XFERRED_DATA);
5546         pkt->pkt_resid = 0;
5547         break;
5548     case MPI2_IOCSTATUS_SCSI_RESIDUAL_MISMATCH:
5549     case MPI2_IOCSTATUS_SCSI_DATA_UNDERRUN:
5550         NDBG31(("data underrun: xferred=%d", xferred));
5551         NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5552         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5553             | STATE_SENT_CMD | STATE_GOT_STATUS);
5554         pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5555         if (pkt->pkt_resid != cmd->cmd_dmacount) {
5556             pkt->pkt_state |= STATE_XFERRED_DATA;
5557         }
5558         break;
5559     case MPI2_IOCSTATUS_SCSI_TASK_TERMINATED:
5560         if (cmd->cmd_active_expiration <= gethrtime()) {
5561             /*
5562              * When timeout requested, propagate
5563              * proper reason and statistics to
5564              * target drivers.
5565              */
5566             mptsas_set_pkt_reason(mpt, cmd, CMD_TIMEOUT,
5567                 STAT_BUS_RESET | STAT_TIMEOUT);
5568         } else {
5569             mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
5570                 STAT_BUS_RESET);

```

```

5571     }
5572     break;
5573     case MPI2_IOCSTATUS_SCSI_IOC_TERMINATED:
5574     case MPI2_IOCSTATUS_SCSI_EXT_TERMINATED:
5575         mptsas_set_pkt_reason(mpt,
5576             cmd, CMD_RESET, STAT_DEV_RESET);
5577         break;
5578     case MPI2_IOCSTATUS_SCSI_IO_DATA_ERROR:
5579     case MPI2_IOCSTATUS_SCSI_PROTOCOL_ERROR:
5580         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET);
5581         mptsas_set_pkt_reason(mpt,
5582             cmd, CMD_TERMINATED, STAT_TERMINATED);
5583         break;
5584     case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
5585     case MPI2_IOCSTATUS_BUSY:
5586         /*
5587          * set throttles to drain
5588          */
5589         for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
5590             ptgt = rehash_next(mpt->m_targets, ptgt)) {
5591             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5592         }

5594         /*
5595          * retry command
5596          */
5597         cmd->cmd_flags |= CFLAG_RETRY;
5598         cmd->cmd_pkt_flags |= FLAG_HEAD;

5600         (void) mptsas_accept_pkt(mpt, cmd);
5601         break;
5602     default:
5603         mptsas_log(mpt, CE_WARN,
5604             "unknown ioc_status = %x\n", ioc_status);
5605         mptsas_log(mpt, CE_CONT, "scsi_state = %x, transfer "
5606             "count = %x, scsi_status = %x", scsi_state,
5607             xferred, scsi_status);
5608         break;
5609     }
5610     break;
5611     case MPI2_SCSI_STATUS_TASK_SET_FULL:
5612         mptsas_handle_qfull(mpt, cmd);
5613         break;
5614     case MPI2_SCSI_STATUS_BUSY:
5615         NDBG31(("scsi_status busy received"));
5616         break;
5617     case MPI2_SCSI_STATUS_RESERVATION_CONFLICT:
5618         NDBG31(("scsi_status reservation conflict received"));
5619         break;
5620     default:
5621         mptsas_log(mpt, CE_WARN, "scsi_status=%x, ioc_status=%x\n",
5622             scsi_status, ioc_status);
5623         mptsas_log(mpt, CE_WARN,
5624             "mptsas_process_intr: invalid scsi status\n");
5625         break;
5626     }
5627 }

5629 static void
5630 mptsas_check_task_mgt(mptsas_t *mpt, pMpi2SCSIManagementReply_t reply,
5631     mptsas_cmd_t *cmd)
5632 {
5633     uint8_t     task_type;
5634     uint16_t    ioc_status;
5635     uint32_t    log_info;
5636     uint16_t    dev_handle;

```

```

5637     struct scsi_pkt *pkt = CMD2PKT(cmd);

5639     task_type = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->TaskType);
5640     ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
5641     log_info = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->IOCLogInfo);
5642     dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->DevHandle);

5644     if (ioc_status != MPI2_IOCSTATUS_SUCCESS) {
5645         mptsas_log(mpt, CE_WARN, "mptsas_check_task_mgt: Task 0x%x "
5646             "failed. IOCStatus=0x%x IOCLogInfo=0x%x target=%d\n",
5647             task_type, ioc_status, log_info, dev_handle);
5648         pkt->pkt_reason = CMD_INCOMPLETE;
5649         return;
5650     }

5652     switch (task_type) {
5653     case MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK:
5654     case MPI2_SCSITASKMGMT_TASKTYPE_CLEAR_TASK_SET:
5655     case MPI2_SCSITASKMGMT_TASKTYPE_QUERY_TASK:
5656     case MPI2_SCSITASKMGMT_TASKTYPE_CLR_ACA:
5657     case MPI2_SCSITASKMGMT_TASKTYPE_ORY_TASK_SET:
5658     case MPI2_SCSITASKMGMT_TASKTYPE_ORY_UNIT_ATTENTION:
5659         break;
5660     case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
5661     case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
5662     case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
5663         /*
5664          * Check for invalid DevHandle of 0 in case application
5665          * sends bad command. DevHandle of 0 could cause problems.
5666          */
5667         if (dev_handle == 0) {
5668             mptsas_log(mpt, CE_WARN, "!Can't flush target with"
5669                 " DevHandle of 0.");
5670         } else {
5671             mptsas_flush_target(mpt, dev_handle, Lun(cmd),
5672                 task_type);
5673         }
5674         break;
5675     default:
5676         mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
5677             task_type);
5678         mptsas_log(mpt, CE_WARN, "ioc status = %x", ioc_status);
5679         break;
5680     }
5681 }

5683 static void
5684 mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg)
5685 {
5686     mptsas_t             *mpt = arg->mpt;
5687     uint64_t             t = arg->t;
5688     mptsas_cmd_t         *cmd;
5689     struct scsi_pkt     *pkt;
5690     mptsas_doneq_thread_list_t *item = &mpt->m_doneq_thread_id[t];

5692     mutex_enter(&item->mutex);
5693     while (item->flag & MPTSAS_DONEQ_THREAD_ACTIVE) {
5694         if (!item->doneq) {
5695             cv_wait(&item->cv, &item->mutex);
5696         }
5697         pkt = NULL;
5698         if ((cmd = mptsas_doneq_thread_rm(mpt, t)) != NULL) {
5699             cmd->cmd_flags |= CFLAG_COMPLETED;
5700             pkt = CMD2PKT(cmd);
5701         }
5702         mutex_exit(&item->mutex);

```

```

5703         if (pkt) {
5704             mptsas_pkt_comp(pkt, cmd);
5705         }
5706         mutex_enter(&item->mutex);
5707     }
5708     mutex_exit(&item->mutex);
5709     mutex_enter(&mpt->m_doneq_mutex);
5710     mpt->m_doneq_thread_n--;
5711     cv_broadcast(&mpt->m_doneq_thread_cv);
5712     mutex_exit(&mpt->m_doneq_mutex);
5713 }

5716 /*
5717  * mpt interrupt handler.
5718  */
5719 static uint_t
5720 mptsas_intr(caddr_t arg1, caddr_t arg2)
5721 {
5722     mptsas_t             *mpt = (void *)arg1;
5723     pMpi2ReplyDescriptorsUnion_t reply_desc_union;
5724     uchar_t             did_reply = FALSE;

5726     NDBG1(("mptsas_intr: arg1 0x%p arg2 0x%p", (void *)arg1, (void *)arg2));

5728     mutex_enter(&mpt->m_mutex);

5730     /*
5731      * If interrupts are shared by two channels then check whether this
5732      * interrupt is genuinely for this channel by making sure first the
5733      * chip is in high power state.
5734      */
5735     if ((mpt->m_options & MPTSAS_OPT_PM) &&
5736         (mpt->m_power_level != PM_LEVEL_D0)) {
5737         mutex_exit(&mpt->m_mutex);
5738         return (DDI_INTR_UNCLAIMED);
5739     }

5741     /*
5742      * If polling, interrupt was triggered by some shared interrupt because
5743      * IOC interrupts are disabled during polling, so polling routine will
5744      * handle any replies. Considering this, if polling is happening,
5745      * return with interrupt unclaimed.
5746      */
5747     if (mpt->m_polled_intr) {
5748         mutex_exit(&mpt->m_mutex);
5749         mptsas_log(mpt, CE_WARN, "mpt_sas: Unclaimed interrupt");
5750         return (DDI_INTR_UNCLAIMED);
5751     }

5753     /*
5754      * Read the istat register.
5755      */
5756     if ((INTPENDING(mpt)) != 0) {
5757         /*
5758          * read fifo until empty.
5759          */
5760 #ifndef __lock_lint
5761         _NOTE(CONSTCOND)
5762 #endif
5763         while (TRUE) {
5764             (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5765                 DDI_DMA_SYNC_FORCPU);
5766             reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
5767                 MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

```

```

5769         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5770             &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5771             ddi_get32(mpt->m_acc_post_queue_hdl,
5772             &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5773             break;
5774         }
5775     }
5776     /*
5777     * The reply is valid, process it according to its
5778     * type. Also, set a flag for updating the reply index
5779     * after they've all been processed.
5780     */
5781     did_reply = TRUE;
5782
5783     mptsas_process_intr(mpt, reply_desc_union);
5784
5785     /*
5786     * Increment post index and roll over if needed.
5787     */
5788     if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5789         mpt->m_post_index = 0;
5790     }
5791 }
5792
5793 /*
5794 * Update the global reply index if at least one reply was
5795 * processed.
5796 */
5797 if (did_reply) {
5798     ddi_put32(mpt->m_datap,
5799         &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
5800 }
5801 } else {
5802     mutex_exit(&mpt->m_mutex);
5803     return (DDI_INTR_UNCLAIMED);
5804 }
5805 NDBG1(("mptsas_intr complete"));
5806
5807 /*
5808 * If no helper threads are created, process the doneq in ISR. If
5809 * helpers are created, use the doneq length as a metric to measure the
5810 * load on the interrupt CPU. If it is long enough, which indicates the
5811 * load is heavy, then we deliver the IO completions to the helpers.
5812 * This measurement has some limitations, although it is simple and
5813 * straightforward and works well for most of the cases at present.
5814 */
5815 if (!mpt->m_doneq_thread_n ||
5816     (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)) {
5817     mptsas_doneq_empty(mpt);
5818 } else {
5819     mptsas_deliver_doneq_thread(mpt);
5820 }
5821
5822 /*
5823 * If there are queued cmd, start them now.
5824 */
5825 if (mpt->m_waitq != NULL) {
5826     mptsas_restart_waitq(mpt);
5827 }
5828
5829 mutex_exit(&mpt->m_mutex);
5830 return (DDI_INTR_CLAIMED);
5831 }
5832
5833 static void
5834 mptsas_process_intr(mptsas_t *mpt,

```

```

5835     pMpi2ReplyDescriptorsUnion_t reply_desc_union)
5836 {
5837     uint8_t reply_type;
5838
5839     ASSERT(mutex_owned(&mpt->m_mutex));
5840
5841     /*
5842     * The reply is valid, process it according to its
5843     * type. Also, set a flag for updated the reply index
5844     * after they've all been processed.
5845     */
5846     reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5847         &reply_desc_union->Default.ReplyFlags);
5848     reply_type &= MPI2_RPY_DESCRIPTOR_FLAGS_TYPE_MASK;
5849     if (reply_type == MPI2_RPY_DESCRIPTOR_FLAGS_SCSI_IO_SUCCESS ||
5850         reply_type == MPI25_RPY_DESCRIPTOR_FLAGS_FAST_PATH_SCSI_IO_SUCCESS) {
5851         mptsas_handle_scsi_io_success(mpt, reply_desc_union);
5852     } else if (reply_type == MPI2_RPY_DESCRIPTOR_FLAGS_ADDRESS_REPLY) {
5853         mptsas_handle_address_reply(mpt, reply_desc_union);
5854     } else {
5855         mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
5856         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5857     }
5858
5859     /*
5860     * Clear the reply descriptor for re-use and increment
5861     * index.
5862     */
5863     ddi_put64(mpt->m_acc_post_queue_hdl,
5864         &((uint64_t *) (void *) mpt->m_post_queue)[mpt->m_post_index],
5865         0xFFFFFFFFFFFFFFFF);
5866     (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5867         DDI_DMA_SYNC_FORDEV);
5868 }
5869
5870 /*
5871 * handle qfull condition
5872 */
5873 static void
5874 mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd)
5875 {
5876     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
5877
5878     if ((++cmd->cmd_qfull_retries > ptgt->m_qfull_retries) ||
5879         (ptgt->m_qfull_retries == 0)) {
5880         /*
5881         * We have exhausted the retries on QFULL, or,
5882         * the target driver has indicated that it
5883         * wants to handle QFULL itself by setting
5884         * qfull-retries capability to 0. In either case
5885         * we want the target driver's QFULL handling
5886         * to kick in. We do this by having pkt_reason
5887         * as CMD_CMPLT and pkt_scbp as STATUS_QFULL.
5888         */
5889         mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5890     } else {
5891         if (ptgt->m_reset_delay == 0) {
5892             ptgt->m_t_throttle =
5893                 max((ptgt->m_t_ncmds - 2), 0);
5894         }
5895
5896         cmd->cmd_pkt_flags |= FLAG_HEAD;
5897         cmd->cmd_flags &= ~(CFLAG_TRANFLAG);
5898         cmd->cmd_flags |= CFLAG_RETRY;
5899
5900         (void) mptsas_accept_pkt(mpt, cmd);

```

```

5902     /*
5903     * when target gives queue full status with no commands
5904     * outstanding (m_t_ncmds == 0), throttle is set to 0
5905     * (HOLD_THROTTLE), and the queue full handling start
5906     * (see psarc/1994/313); if there are commands outstanding,
5907     * throttle is set to (m_t_ncmds - 2)
5908     */
5909     if (ptgt->m_t_throttle == HOLD_THROTTLE) {
5910         /*
5911         * By setting throttle to QFULL_THROTTLE, we
5912         * avoid submitting new commands and in
5913         * mptsas_restart_cmd find out slots which need
5914         * their throttles to be cleared.
5915         */
5916         mptsas_set_throttle(mpt, ptgt, QFULL_THROTTLE);
5917         if (mpt->m_restart_cmd_timeid == 0) {
5918             mpt->m_restart_cmd_timeid =
5919                 timeout(mptsas_restart_cmd, mpt,
5920                     ptgt->m_qfull_retry_interval);
5921         }
5922     }
5923 }
5924 }

5926 mptsas_phymask_t
5927 mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport)
5928 {
5929     mptsas_phymask_t    phy_mask = 0;
5930     uint8_t             i = 0;

5932     NDBG20(("mptsas%d physport_to_phymask enter", mpt->m_instance));

5934     ASSERT(mutex_owned(&mpt->m_mutex));

5936     /*
5937     * If physport is 0xFF, this is a RAID volume. Use phymask of 0.
5938     */
5939     if (physport == 0xFF) {
5940         return (0);
5941     }

5943     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
5944         if (mpt->m_phy_info[i].attached_devhdl &&
5945             (mpt->m_phy_info[i].phy_mask != 0) &&
5946             (mpt->m_phy_info[i].port_num == physport)) {
5947             phy_mask = mpt->m_phy_info[i].phy_mask;
5948             break;
5949         }
5950     }
5951     NDBG20(("mptsas%d physport_to_phymask:physport :%x phymask :%x, ",
5952         mpt->m_instance, physport, phy_mask));
5953     return (phy_mask);
5954 }

5956 /*
5957 * mpt free device handle after device gone, by use of passthrough
5958 */
5959 static int
5960 mptsas_free_devhdl(mptsas_t *mpt, uint16_t devhdl)
5961 {
5962     Mpi2SasIoUnitControlRequest_t    req;
5963     Mpi2SasIoUnitControlReply_t      rep;
5964     int                                ret;

5966     ASSERT(mutex_owned(&mpt->m_mutex));

```

```

5968     /*
5969     * Need to compose a SAS IO Unit Control request message
5970     * and call mptsas_do_passthru() function
5971     */
5972     bzero(&req, sizeof (req));
5973     bzero(&rep, sizeof (rep));

5975     req.Function = MPI2_FUNCTION_SAS_IO_UNIT_CONTROL;
5976     req.Operation = MPI2_SAS_OP_REMOVE_DEVICE;
5977     req.DevHandle = LE_16(devhdl);

5979     ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
5980         sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
5981     if (ret != 0) {
5982         cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
5983             "Control error %d", ret);
5984         return (DDI_FAILURE);
5985     }

5987     /* do passthrough success, check the ioc status */
5988     if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
5989         cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
5990             "Control IOCStatus %d", LE_16(rep.IOCStatus));
5991         return (DDI_FAILURE);
5992     }

5994     return (DDI_SUCCESS);
5995 }

5997 static void
5998 mptsas_update_phymask(mptsas_t *mpt)
5999 {
6000     mptsas_phymask_t mask = 0, phy_mask;
6001     char                *phy_mask_name;
6002     uint8_t             current_port;
6003     int                 i, j;

6005     NDBG20(("mptsas%d update phymask ", mpt->m_instance));

6007     ASSERT(mutex_owned(&mpt->m_mutex));

6009     (void) mptsas_get_sas_io_unit_page(mpt);

6011     phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);

6013     for (i = 0; i < mpt->m_num_phys; i++) {
6014         phy_mask = 0x00;

6016         if (mpt->m_phy_info[i].attached_devhdl == 0)
6017             continue;

6019         bzero(phy_mask_name, sizeof (phy_mask_name));

6021         current_port = mpt->m_phy_info[i].port_num;

6023         if ((mask & (1 << i)) != 0)
6024             continue;

6026         for (j = 0; j < mpt->m_num_phys; j++) {
6027             if (mpt->m_phy_info[j].attached_devhdl &&
6028                 (mpt->m_phy_info[j].port_num == current_port)) {
6029                 phy_mask |= (1 << j);
6030             }
6031         }
6032         mask = mask | phy_mask;

```

```

6034         for (j = 0; j < mpt->m_num_phys; j++) {
6035             if ((phy_mask >> j) & 0x01) {
6036                 mpt->m_phy_info[j].phy_mask = phy_mask;
6037             }
6038         }
6040         (void) sprintf(phy_mask_name, "%x", phy_mask);
6042         mutex_exit(&mpt->m_mutex);
6043         /*
6044          * register a iport, if the port has already been existed
6045          * SCSA will do nothing and just return.
6046          */
6047         (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
6048         mutex_enter(&mpt->m_mutex);
6049     }
6050     kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
6051     NDBG20(("mptsas%d update phymask return", mpt->m_instance));
6052 }
6054 /*
6055  * mptsas_handle_dr is a task handler for DR, the DR action includes:
6056  * 1. Directly attached Device Added/Removed.
6057  * 2. Expander Device Added/Removed.
6058  * 3. Indirectly Attached Device Added/Expander.
6059  * 4. LUNs of a existing device status change.
6060  * 5. RAID volume created/deleted.
6061  * 6. Member of RAID volume is released because of RAID deletion.
6062  * 7. Physical disks are removed because of RAID creation.
6063  */
6064 static void
6065 mptsas_handle_dr(void *args) {
6066     mptsas_topo_change_list_t    *topo_node = NULL;
6067     mptsas_topo_change_list_t    *save_node = NULL;
6068     mptsas_t                     *mpt;
6069     dev_info_t                   *parent = NULL;
6070     mptsas_phymask_t             phymask = 0;
6071     char                         *phy_mask_name;
6072     uint8_t                      flags = 0, physport = 0xff;
6073     uint8_t                      port_update = 0;
6074     uint_t                       event;
6076     topo_node = (mptsas_topo_change_list_t *)args;
6078     mpt = topo_node->mpt;
6079     event = topo_node->event;
6080     flags = topo_node->flags;
6082     phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
6084     NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));
6086     switch (event) {
6087     case MPTSAS_DR_EVENT_RECONFIG_TARGET:
6088         if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6089             (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
6090             (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6091             /*
6092              * Direct attached or expander attached device added
6093              * into system or a Phys Disk that is being unhidden.
6094              */
6095             port_update = 1;
6096         }
6097         break;
6098     case MPTSAS_DR_EVENT_RECONFIG_SMP:

```

```

6099         /*
6100          * New expander added into system, it must be the head
6101          * of topo_change_list_t
6102          */
6103         port_update = 1;
6104         break;
6105     default:
6106         port_update = 0;
6107         break;
6108     }
6109     /*
6110     * All cases port_update == 1 may cause initiator port form change
6111     */
6112     mutex_enter(&mpt->m_mutex);
6113     if (mpt->m_port_chng && port_update) {
6114         /*
6115          * mpt->m_port_chng flag indicates some PHYs of initiator
6116          * port have changed to online. So when expander added or
6117          * directly attached device online event come, we force to
6118          * update port information by issuing SAS IO Unit Page and
6119          * update PHYMASKs.
6120          */
6121         (void) mptsas_update_phymask(mpt);
6122         mpt->m_port_chng = 0;
6124     }
6125     mutex_exit(&mpt->m_mutex);
6126     while (topo_node) {
6127         phymask = 0;
6128         if (parent == NULL) {
6129             physport = topo_node->un.physport;
6130             event = topo_node->event;
6131             flags = topo_node->flags;
6132             if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
6133                 MPTSAS_DR_EVENT_OFFLINE_SMP)) {
6134                 /*
6135                  * For all offline events, phymask is known
6136                  */
6137                 phymask = topo_node->un.phymask;
6138                 goto find_parent;
6139             }
6140             if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6141                 goto handle_topo_change;
6142             }
6143             if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
6144                 phymask = topo_node->un.phymask;
6145                 goto find_parent;
6146             }
6148             if ((flags ==
6149                 MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
6150                 (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
6151                 /*
6152                  * There is no any field in IR_CONFIG_CHANGE
6153                  * event indicate physport/phynum, let's get
6154                  * parent after SAS Device Page0 request.
6155                  */
6156                 goto handle_topo_change;
6157             }
6159             mutex_enter(&mpt->m_mutex);
6160             if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6161                 /*
6162                  * If the direct attached device added or a
6163                  * phys disk is being unhidden, argument
6164                  * physport actually is PHY#, so we have to get

```

```

6165         * phymask according PHY#.
6166         */
6167         physport = mpt->m_phy_info[physport].port_num;
6168     }
6170     /*
6171     * Translate physport to phymask so that we can search
6172     * parent dip.
6173     */
6174     phymask = mptsas_physport_to_phymask(mpt,
6175     physport);
6176     mutex_exit(&mpt->m_mutex);

6178 find_parent:
6179     bzero(phy_mask_name, MPTSAS_MAX_PHYS);
6180     /*
6181     * For RAID topology change node, write the iport name
6182     * as v0.
6183     */
6184     if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
6185         (void) sprintf(phy_mask_name, "v0");
6186     } else {
6187         /*
6188         * phymask can be 0 if the drive has been
6189         * pulled by the time an add event is
6190         * processed. If phymask is 0, just skip this
6191         * event and continue.
6192         */
6193         if (phymask == 0) {
6194             mutex_enter(&mpt->m_mutex);
6195             save_node = topo_node;
6196             topo_node = topo_node->next;
6197             ASSERT(save_node);
6198             kmem_free(save_node,
6199             sizeof(mptsas_topo_change_list_t));
6200             mutex_exit(&mpt->m_mutex);

6202             parent = NULL;
6203             continue;
6204         }
6205         (void) sprintf(phy_mask_name, "%x", phymask);
6206     }
6207     parent = scsi_hba_iport_find(mpt->m_dip,
6208     phy_mask_name);
6209     if (parent == NULL) {
6210         mptsas_log(mpt, CE_WARN, "Failed to find an "
6211         "iport, should not happen!");
6212         goto out;
6213     }

6215     }
6216     ASSERT(parent);
6217 handle_topo_change:

6219     mutex_enter(&mpt->m_mutex);
6220     /*
6221     * If HBA is being reset, don't perform operations depending
6222     * on the IOC. We must free the topo list, however.
6223     */
6224     if (!mpt->m_in_reset)
6225         mptsas_handle_topo_change(topo_node, parent);
6226     else
6227         NDBG20(("skipping topo change received during reset"));
6228     save_node = topo_node;
6229     topo_node = topo_node->next;
6230     ASSERT(save_node);

```

```

6231     kmem_free(save_node, sizeof(mptsas_topo_change_list_t));
6232     mutex_exit(&mpt->m_mutex);

6234     if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6235     (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
6236     (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) {
6237         /*
6238         * If direct attached device associated, make sure
6239         * reset the parent before start the next one. But
6240         * all devices associated with expander shares the
6241         * parent. Also, reset parent if this is for RAID.
6242         */
6243         parent = NULL;
6244     }
6245     }
6246 out:
6247     kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
6248 }

6250 static void
6251 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
6252     dev_info_t *parent)
6253 {
6254     mptsas_target_t *ptgt = NULL;
6255     mptsas_smp_t *psmp = NULL;
6256     mptsas_t *mpt = (void *)topo_node->mpt;
6257     uint16_t devhdl;
6258     uint16_t attached_devhdl;
6259     uint64_t sas_wwn = 0;
6260     int rval = 0;
6261     uint32_t page_address;
6262     uint8_t phy, flags;
6263     char *addr = NULL;
6264     dev_info_t *lundip;
6265     int circ = 0, circ1 = 0;
6266     char attached_wwnstr[MPTSAS_WWN_STRLEN];

6268     NDBG20(("mptsas%d handle_topo_change enter, devhdl 0x%x,"
6269     "event 0x%x, flags 0x%x", mpt->m_instance, topo_node->devhdl,
6270     topo_node->event, topo_node->flags));

6272     ASSERT(mutex_owned(&mpt->m_mutex));

6274     switch (topo_node->event) {
6275     case MPTSAS_DR_EVENT_RECONFIG_TARGET:
6276     {
6277         char *phy_mask_name;
6278         mptsas_phymask_t phymask = 0;

6280         if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
6281             /*
6282             * Get latest RAID info.
6283             */
6284             (void) mptsas_get_raid_info(mpt);
6285             ptgt = rehash_linear_search(mpt->m_targets,
6286             mptsas_target_eval_devhdl, &topo_node->devhdl);
6287             if (ptgt == NULL)
6288                 break;
6289         } else {
6290             ptgt = (void *)topo_node->object;
6291         }

6293         if (ptgt == NULL) {
6294             /*
6295             * If a Phys Disk was deleted, RAID info needs to be
6296             * updated to reflect the new topology.

```



```

6297     */
6298     (void) mptsas_get_raid_info(mpt);
6300     /*
6301     * Get sas device page 0 by DevHandle to make sure if
6302     * SSP/SATA end device exist.
6303     */
6304     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
6305     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
6306     topo_node->devhdl;
6308     rval = mptsas_get_target_device_info(mpt, page_address,
6309     &devhdl, &ptgt);
6310     if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
6311         mptsas_log(mpt, CE_NOTE,
6312         "mptsas handle topo change: target %d is "
6313         "not a SAS/SATA device. \n",
6314         topo_node->devhdl);
6315     } else if (rval == DEV_INFO_FAIL_ALLOC) {
6316         mptsas_log(mpt, CE_NOTE,
6317         "mptsas handle topo change: could not "
6318         "allocate memory. \n");
6319     }
6320     /*
6321     * If rval is DEV_INFO_PHYS_DISK than there is nothing
6322     * else to do, just leave.
6323     */
6324     if (rval != DEV_INFO_SUCCESS) {
6325         return;
6326     }
6327 }
6329 ASSERT(ptgt->m_devhdl == topo_node->devhdl);
6331 mutex_exit(&mpt->m_mutex);
6332 flags = topo_node->flags;
6334 if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
6335     phymask = ptgt->m_addr.mta_phymask;
6336     phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
6337     (void) sprintf(phy_mask_name, "%x", phymask);
6338     parent = scsi_hba_iport_find(mpt->m_dip,
6339     phy_mask_name);
6340     kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
6341     if (parent == NULL) {
6342         mptsas_log(mpt, CE_WARN, "Failed to find a "
6343         "iport for PD, should not happen!");
6344         mutex_enter(&mpt->m_mutex);
6345         break;
6346     }
6347 }
6349 if (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
6350     ndi_devi_enter(parent, &circl);
6351     (void) mptsas_config_raid(parent, topo_node->devhdl,
6352     &lundip);
6353     ndi_devi_exit(parent, circl);
6354 } else {
6355     /*
6356     * hold nexus for bus configure
6357     */
6358     ndi_devi_enter(scsi_vhci_dip, &circ);
6359     ndi_devi_enter(parent, &circl);
6360     rval = mptsas_config_target(parent, ptgt);
6361     /*
6362     * release nexus for bus configure

```

```

6363     */
6364     ndi_devi_exit(parent, circl);
6365     ndi_devi_exit(scsi_vhci_dip, circ);
6367     /*
6368     * Add parent's props for SMHBA support
6369     */
6370     if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6371         bzero(attached_wwnstr,
6372         sizeof(attached_wwnstr));
6373         (void) sprintf(attached_wwnstr, "%w%016"PRIx64,
6374         ptgt->m_addr.mta_wwn);
6375         if (ddi_prop_update_string(DDI_DEV_T_NONE,
6376         parent,
6377         SCSI_ADDR_PROP_ATTACHED_PORT,
6378         attached_wwnstr)
6379         != DDI_PROP_SUCCESS) {
6380             (void) ddi_prop_remove(DDI_DEV_T_NONE,
6381             parent,
6382             SCSI_ADDR_PROP_ATTACHED_PORT);
6383             mptsas_log(mpt, CE_WARN, "Failed to "
6384             "attached-port props");
6385             return;
6386         }
6387         if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6388         MPTSAS_NUM_PHYS, 1) !=
6389         DDI_PROP_SUCCESS) {
6390             (void) ddi_prop_remove(DDI_DEV_T_NONE,
6391             parent, MPTSAS_NUM_PHYS);
6392             mptsas_log(mpt, CE_WARN, "Failed to "
6393             "create num-phys props");
6394             return;
6395         }
6397     /*
6398     * Update PHY info for smhba
6399     */
6400     mutex_enter(&mpt->m_mutex);
6401     if (mptsas_smhba_phy_init(mpt)) {
6402         mutex_exit(&mpt->m_mutex);
6403         mptsas_log(mpt, CE_WARN, "mptsas phy"
6404         " update failed");
6405         return;
6406     }
6407     mutex_exit(&mpt->m_mutex);
6409     /*
6410     * topo_node->un.physport is really the PHY#
6411     * for direct attached devices
6412     */
6413     mptsas_smhba_set_one_phy_props(mpt, parent,
6414     topo_node->un.physport, &attached_devhdl);
6416     if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6417     MPTSAS_VIRTUAL_PORT, 0) !=
6418     DDI_PROP_SUCCESS) {
6419         (void) ddi_prop_remove(DDI_DEV_T_NONE,
6420         parent, MPTSAS_VIRTUAL_PORT);
6421         mptsas_log(mpt, CE_WARN,
6422         "mptsas virtual-port "
6423         "port prop update failed");
6424         return;
6425     }
6426     }
6427 }
6428     mutex_enter(&mpt->m_mutex);

```

```

6430         NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
6431             "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
6432             ptgt->m_addr.mta_phymask));
6433         break;
6434     }
6435     case MPTSAS_DR_EVENT_OFFLINE_TARGET:
6436     {
6437         devhdl = topo_node->devhdl;
6438         ptgt = rehash_linear_search(mpt->m_targets,
6439             mptsas_target_eval_devhdl, &devhdl);
6440         if (ptgt == NULL)
6441             break;

6443         sas_wnn = ptgt->m_addr.mta_wnn;
6444         phy = ptgt->m_phynum;

6446         addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

6448         if (sas_wnn) {
6449             (void) sprintf(addr, "w%016"PRIx64, sas_wnn);
6450         } else {
6451             (void) sprintf(addr, "p%x", phy);
6452         }
6453         ASSERT(ptgt->m_devhdl == devhdl);

6455         if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
6456             (topo_node->flags ==
6457             MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6458             /*
6459              * Get latest RAID info if RAID volume status changes
6460              * or Phys Disk status changes
6461              */
6462             (void) mptsas_get_raid_info(mpt);
6463         }
6464         /*
6465          * Abort all outstanding command on the device
6466          */
6467         rval = mptsas_do_scsi_reset(mpt, devhdl);
6468         if (rval) {
6469             NDBG20(("mptsas%d handle_topo_change to reset target "
6470                 "before offline devhdl:%x, phymask:%x, rval:%x",
6471                 mpt->m_instance, ptgt->m_devhdl,
6472                 ptgt->m_addr.mta_phymask, rval));
6473         }

6475         mutex_exit(&mpt->m_mutex);

6477         ndi_devi_enter(scsi_vhci_dip, &circ);
6478         ndi_devi_enter(parent, &circ1);
6479         rval = mptsas_offline_target(parent, addr);
6480         ndi_devi_exit(parent, circ1);
6481         ndi_devi_exit(scsi_vhci_dip, circ);
6482         NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6483             "phymask:%x, rval:%x", mpt->m_instance,
6484             ptgt->m_devhdl, ptgt->m_addr.mta_phymask, rval));

6486         kmem_free(addr, SCSI_MAXNAMELEN);

6488         /*
6489          * Clear parent's props for SMHBA support
6490          */
6491         flags = topo_node->flags;
6492         if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6493             bzero(attached_wnnstr, sizeof (attached_wnnstr));
6494             if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,

```

```

6495         SCSI_ADDR_PROP_ATTACHED_PORT, attached_wnnstr) !=
6496         DDI_PROP_SUCCESS) {
6497             (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6498                 SCSI_ADDR_PROP_ATTACHED_PORT);
6499             mptsas_log(mpt, CE_WARN, "mptsas attached port "
6500                 "prop update failed");
6501             break;
6502         }
6503         if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6504             MPTSAS_NUM_PHYS, 0) !=
6505             DDI_PROP_SUCCESS) {
6506             (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6507                 MPTSAS_NUM_PHYS);
6508             mptsas_log(mpt, CE_WARN, "mptsas num phys "
6509                 "prop update failed");
6510             break;
6511         }
6512         if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6513             MPTSAS_VIRTUAL_PORT, 1) !=
6514             DDI_PROP_SUCCESS) {
6515             (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6516                 MPTSAS_VIRTUAL_PORT);
6517             mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6518                 "prop update failed");
6519             break;
6520         }
6521     }

6523     mutex_enter(&mpt->m_mutex);
6524     ptgt->m_led_status = 0;
6525     (void) mptsas_flush_led_status(mpt, ptgt);
6526     if (rval == DDI_SUCCESS) {
6527         rehash_remove(mpt->m_targets, ptgt);
6528         ptgt = NULL;
6529     } else {
6530         /*
6531          * clean DR_INTRANSITION flag to allow I/O down to
6532          * PHCI driver since failover finished.
6533          * Invalidate the devhdl
6534          */
6535         ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6536         ptgt->m_tgt_unconfigured = 0;
6537         mutex_enter(&mpt->m_tx_waitq_mutex);
6538         ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
6539         mutex_exit(&mpt->m_tx_waitq_mutex);
6540     }

6542     /*
6543      * Send SAS IO Unit Control to free the dev handle
6544      */
6545     if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6546         (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
6547         rval = mptsas_free_devhdl(mpt, devhdl);

6549         NDBG20(("mptsas%d handle_topo_change to remove "
6550             "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6551             rval));
6552     }

6554     break;
6555 }
6556 case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
6557 {
6558     devhdl = topo_node->devhdl;
6559     /*
6560      * If this is the remove handle event, do a reset first.

```

```

6561     */
6562     if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6563         rval = mptsas_do_scsi_reset(mpt, devhdl);
6564         if (rval) {
6565             NDBG20(("mpt%d reset target before remove "
6566                  "devhdl:%x, rval:%x", mpt->m_instance,
6567                  devhdl, rval));
6568         }
6569     }
6571     /*
6572     * Send SAS IO Unit Control to free the dev handle
6573     */
6574     rval = mptsas_free_devhdl(mpt, devhdl);
6575     NDBG20(("mptsas%d handle_topo_change to remove "
6576            "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6577            rval));
6578     break;
6579 }
6580 case MPTSAS_DR_EVENT_RECONFIG_SMP:
6581 {
6582     mptsas_smp_t smp;
6583     dev_info_t *smpdip;
6585     devhdl = topo_node->devhdl;
6587     page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
6588                    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
6589     rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
6590     if (rval != DDI_SUCCESS) {
6591         mptsas_log(mpt, CE_WARN, "failed to online smp, "
6592                  "handle %x", devhdl);
6593         return;
6594     }
6596     psmpt = mptsas_smp_alloc(mpt, &smp);
6597     if (psmpt == NULL) {
6598         return;
6599     }
6601     mutex_exit(&mpt->m_mutex);
6602     ndi_devi_enter(parent, &circl);
6603     (void) mptsas_online_smp(parent, psmpt, &smpdip);
6604     ndi_devi_exit(parent, circl);
6606     mutex_enter(&mpt->m_mutex);
6607     break;
6608 }
6609 case MPTSAS_DR_EVENT_OFFLINE_SMP:
6610 {
6611     devhdl = topo_node->devhdl;
6612     uint32_t dev_info;
6614     psmpt = rehash_linear_search(mpt->m_smp_targets,
6615                                 mptsas_smp_eval_devhdl, &devhdl);
6616     if (psmpt == NULL)
6617         break;
6618     /*
6619     * The mptsas_smp_t data is warned only if the dip is offlined
6620     * successfully.
6621     */
6622     mutex_exit(&mpt->m_mutex);
6624     ndi_devi_enter(parent, &circl);
6625     rval = mptsas_offline_smp(parent, psmpt, NDI_DEVI_REMOVE);
6626     ndi_devi_exit(parent, circl);

```

```

6628     dev_info = psmpt->m_deviceinfo;
6629     if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
6630         DEVINFO_DIRECT_ATTACHED) {
6631         if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6632                                MPTSAS_VIRTUAL_PORT, 1) !=
6633             DDI_PROP_SUCCESS) {
6634             (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6635                                   MPTSAS_VIRTUAL_PORT);
6636             mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6637                       "prop update failed");
6638             return;
6639         }
6640     }
6641     /*
6642     * Check whether the smp connected to the iport,
6643     */
6644     if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6645                            MPTSAS_NUM_PHYS, 0) !=
6646         DDI_PROP_SUCCESS) {
6647         (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6648                               MPTSAS_NUM_PHYS);
6649         mptsas_log(mpt, CE_WARN, "mptsas num phys "
6650                   "prop update failed");
6651         return;
6652     }
6653     /*
6654     * Clear parent's attached-port props
6655     */
6656     bzero(attached_wnvstr, sizeof (attached_wnvstr));
6657     if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6658                               SCSI_ADDR_PROP_ATTACHED_PORT, attached_wnvstr) !=
6659         DDI_PROP_SUCCESS) {
6660         (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6661                               SCSI_ADDR_PROP_ATTACHED_PORT);
6662         mptsas_log(mpt, CE_WARN, "mptsas attached port "
6663                   "prop update failed");
6664         return;
6665     }
6667     mutex_enter(&mpt->m_mutex);
6668     NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6669            "rval:%x", mpt->m_instance, psmpt->m_devhdl, rval));
6670     if (rval == DDI_SUCCESS) {
6671         rehash_remove(mpt->m_smp_targets, psmpt);
6672     } else {
6673         psmpt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6674     }
6676     bzero(attached_wnvstr, sizeof (attached_wnvstr));
6678     break;
6679 }
6680 default:
6681     return;
6682 }
6683 }
6685 /*
6686 * Record the event if its type is enabled in mpt instance by ioctl.
6687 */
6688 static void
6689 mptsas_record_event(void *args)
6690 {
6691     m_replyh_arg_t *replyh_arg;
6692     pMpi2EventNotificationReply_t eventreply;

```

```

6693     uint32_t          event, rfm;
6694     mptsas_t          *mpt;
6695     int               i, j;
6696     uint16_t         event_data_len;
6697     boolean_t        sendAEN = FALSE;

6699     replyh_arg = (m_replyh_arg_t *)args;
6700     rfm = replyh_arg->rfm;
6701     mpt = replyh_arg->mpt;

6703     eventreply = (pMpi2EventNotificationReply_t)
6704     (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
6705     event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6708     /*
6709     * Generate a system event to let anyone who cares know that a
6710     * LOG_ENTRY_ADDED event has occurred. This is sent no matter what the
6711     * event mask is set to.
6712     */
6713     if (event == MPI2_EVENT_LOG_ENTRY_ADDED) {
6714         sendAEN = TRUE;
6715     }

6717     /*
6718     * Record the event only if it is not masked. Determine which dword
6719     * and bit of event mask to test.
6720     */
6721     i = (uint8_t)(event / 32);
6722     j = (uint8_t)(event % 32);
6723     if ((i < 4) && ((1 << j) & mpt->m_event_mask[i])) {
6724         i = mpt->m_event_index;
6725         mpt->m_events[i].Type = event;
6726         mpt->m_events[i].Number = ++mpt->m_event_number;
6727         bzero(mpt->m_events[i].Data, MPTSAS_MAX_EVENT_DATA_LENGTH * 4);
6728         event_data_len = ddi_get16(mpt->m_acc_reply_frame_hdl,
6729             &eventreply->EventDataLength);

6731         if (event_data_len > 0) {
6732             /*
6733             * Limit data to size in m_event entry
6734             */
6735             if (event_data_len > MPTSAS_MAX_EVENT_DATA_LENGTH) {
6736                 event_data_len = MPTSAS_MAX_EVENT_DATA_LENGTH;
6737             }
6738             for (j = 0; j < event_data_len; j++) {
6739                 mpt->m_events[i].Data[j] =
6740                 ddi_get32(mpt->m_acc_reply_frame_hdl,
6741                     &(eventreply->EventData[j]));
6742             }

6744             /*
6745             * check for index wrap-around
6746             */
6747             if (++i == MPTSAS_EVENT_QUEUE_SIZE) {
6748                 i = 0;
6749             }
6750             mpt->m_event_index = (uint8_t)i;

6752             /*
6753             * Set flag to send the event.
6754             */
6755             sendAEN = TRUE;
6756         }
6757     }

```

```

6759     /*
6760     * Generate a system event if flag is set to let anyone who cares know
6761     * that an event has occurred.
6762     */
6763     if (sendAEN) {
6764         (void) ddi_log_sysevent(mpt->m_dip, DDI_VENDOR_LSI, "MPT_SAS",
6765             "SAS", NULL, NULL, DDI_NOSLEEP);
6766     }
6767 }

6769 #define SMP_RESET_IN_PROGRESS MPI2_EVENT_SAS_TOPO_LR_SMP_RESET_IN_PROGRESS
6770 /*
6771 * handle sync events from ioc in interrupt
6772 * return value:
6773 * DDI_SUCCESS: The event is handled by this func
6774 * DDI_FAILURE: Event is not handled
6775 */
6776 static int
6777 mptsas_handle_event_sync(void *args)
6778 {
6779     m_replyh_arg_t          *replyh_arg;
6780     pMpi2EventNotificationReply_t eventreply;
6781     uint32_t                event, rfm;
6782     mptsas_t                *mpt;
6783     uint_t                  iocstatus;

6785     replyh_arg = (m_replyh_arg_t *)args;
6786     rfm = replyh_arg->rfm;
6787     mpt = replyh_arg->mpt;

6789     ASSERT(mutex_owned(&mpt->m_mutex));

6791     eventreply = (pMpi2EventNotificationReply_t)
6792     (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
6793     event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6795     if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
6796         &eventreply->IOCStatus)) {
6797         if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
6798             mptsas_log(mpt, CE_WARN,
6799                 "!mptsas_handle_event_sync: event 0x%x, "
6800                 "IOCStatus=0x%x, "
6801                 "IOCLogInfo=0x%x", event, iocstatus,
6802                 ddi_get32(mpt->m_acc_reply_frame_hdl,
6803                     &eventreply->IOCLogInfo));
6804         } else {
6805             mptsas_log(mpt, CE_WARN,
6806                 "mptsas_handle_event_sync: event 0x%x, "
6807                 "IOCStatus=0x%x, "
6808                 "(IOCLogInfo=0x%x)", event, iocstatus,
6809                 ddi_get32(mpt->m_acc_reply_frame_hdl,
6810                     &eventreply->IOCLogInfo));
6811         }
6812     }

6814     /*
6815     * figure out what kind of event we got and handle accordingly
6816     */
6817     switch (event) {
6818     case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
6819     {
6820         pMpi2EventDataSasTopologyChangeList_t sas_topo_change_list;
6821         uint8_t num_entries, expstatus, phy;
6822         uint8_t phystatus, physport, state, i;
6823         uint8_t start_phy_num, link_rate;
6824         uint16_t dev_handle, reason_code;

```



```

6957         "failed speed negotiation");
6958         break;
6959     case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6960         (void) sprintf(curr, "SATA OOB "
6961             "complete");
6962         break;
6963     case SMP_RESET_IN_PROGRESS:
6964         (void) sprintf(curr, "SMP reset in "
6965             "progress");
6966         break;
6967     case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6968         (void) sprintf(curr, "is online at "
6969             "1.5 Gbps");
6970         break;
6971     case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6972         (void) sprintf(curr, "is online at 3.0 "
6973             "Gbps");
6974         break;
6975     case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6976         (void) sprintf(curr, "is online at 6.0 "
6977             "Gbps");
6978         break;
6979     case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
6980         (void) sprintf(curr,
6981             "is online at 12.0 Gbps");
6982         break;
6983     default:
6984         (void) sprintf(curr, "state is "
6985             "unknown");
6986         break;
6987     }
6988     /*
6989     * New target device added into the system.
6990     * Set association flag according to if an
6991     * expander is used or not.
6992     */
6993     exp_flag =
6994         MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
6995     if (flags ==
6996         MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
6997         flags = exp_flag;
6998     }
6999     topo_node = kmem_zalloc(
7000         sizeof (mptsas_topo_change_list_t),
7001         KM_SLEEP);
7002     topo_node->mpt = mpt;
7003     topo_node->event =
7004         MPTSAS_DR_EVENT_RECONFIG_TARGET;
7005     if (expd_handle == 0) {
7006         /*
7007         * Per MPI 2, if expander dev handle
7008         * is 0, it's a directly attached
7009         * device. So driver use PHY to decide
7010         * which iport is associated
7011         */
7012         physport = phy;
7013         mpt->m_port_chng = 1;
7014     }
7015     topo_node->un.physport = physport;
7016     topo_node->devhdl = dev_handle;
7017     topo_node->flags = flags;
7018     topo_node->object = NULL;
7019     if (topo_head == NULL) {
7020         topo_head = topo_tail = topo_node;
7021     } else {
7022         topo_tail->next = topo_node;

```

```

7023         topo_tail = topo_node;
7024     }
7025     break;
7026 }
7027 case MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING:
7028 {
7029     NDBG20(("mptsas%d phy %d physical_port %d "
7030         "dev_handle %d removed", mpt->m_instance,
7031         phy, physport, dev_handle));
7032     /*
7033     * Set association flag according to if an
7034     * expander is used or not.
7035     */
7036     exp_flag =
7037         MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
7038     if (flags ==
7039         MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
7040         flags = exp_flag;
7041     }
7042     /*
7043     * Target device is removed from the system
7044     * Before the device is really offline from
7045     * from system.
7046     */
7047     ptgt = rehash_linear_search(mpt->m_targets,
7048         mptsas_target_eval_devhdl, &dev_handle);
7049     /*
7050     * If ptgt is NULL here, it means that the
7051     * DevHandle is not in the hash table. This is
7052     * reasonable sometimes. For example, if a
7053     * disk was pulled, then added, then pulled
7054     * again, the disk will not have been put into
7055     * the hash table because the add event will
7056     * have an invalid phymask. BUT, this does not
7057     * mean that the DevHandle is invalid. The
7058     * controller will still have a valid DevHandle
7059     * that must be removed. To do this, use the
7060     * MPTSAS_TOPO_FLAG_REMOVE_HANDLE event.
7061     */
7062     if (ptgt == NULL) {
7063         topo_node = kmem_zalloc(
7064             sizeof (mptsas_topo_change_list_t),
7065             KM_SLEEP);
7066         topo_node->mpt = mpt;
7067         topo_node->un.phymask = 0;
7068         topo_node->event =
7069             MPTSAS_TOPO_FLAG_REMOVE_HANDLE;
7070         topo_node->devhdl = dev_handle;
7071         topo_node->flags = flags;
7072         topo_node->object = NULL;
7073         if (topo_head == NULL) {
7074             topo_head = topo_tail =
7075                 topo_node;
7076         } else {
7077             topo_tail->next = topo_node;
7078             topo_tail = topo_node;
7079         }
7080         break;
7081     }
7082     /*
7083     * Update DR flag immediately avoid I/O failure
7084     * before failover finish. Pay attention to the
7085     * mutex protect, we need grab m_tx_waitq_mutex
7086     * during set m_dr_flag because we won't add
7087     * the following command into waitq, instead,
7088

```

```

7089     * we need return TRAN_BUSY in the tran_start
7090     * context.
7091     */
7092     mutex_enter(&mpt->m_tx_waitq_mutex);
7093     ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
7094     mutex_exit(&mpt->m_tx_waitq_mutex);

7096     topo_node = kmem_zalloc(
7097         sizeof (mptsas_topo_change_list_t),
7098         KM_SLEEP);
7099     topo_node->mpt = mpt;
7100     topo_node->un.phymask =
7101         ptgt->m_addr.mta_phymask;
7102     topo_node->event =
7103         MPTSAS_DR_EVENT_OFFLINE_TARGET;
7104     topo_node->devhdl = dev_handle;
7105     topo_node->flags = flags;
7106     topo_node->object = NULL;
7107     if (topo_head == NULL) {
7108         topo_head = topo_tail = topo_node;
7109     } else {
7110         topo_tail->next = topo_node;
7111         topo_tail = topo_node;
7112     }
7113     break;
7114 }
7115 case MPI2_EVENT_SAS_TOPO_RC_PHY_CHANGED:
7116     link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
7117         &sas_topo_change_list->PHY[i].LinkRate);
7118     state = (link_rate &
7119         MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
7120         MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
7121     pSmhba = &mpt->m_phy_info[i].smhba_info;
7122     pSmhba->negotiated_link_rate = state;
7123     switch (state) {
7124     case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
7125         (void) sprintf(curr, "is disabled");
7126         mptsas_smhba_log_sysevent(mpt,
7127             ESC_SAS_PHY_EVENT,
7128             SAS_PHY_REMOVE,
7129             &mpt->m_phy_info[i].smhba_info);
7130         mpt->m_phy_info[i].smhba_info.
7131             negotiated_link_rate
7132             = 0x1;
7133         break;
7134     case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
7135         (void) sprintf(curr, "is offline, "
7136             "failed speed negotiation");
7137         mptsas_smhba_log_sysevent(mpt,
7138             ESC_SAS_PHY_EVENT,
7139             SAS_PHY_OFFLINE,
7140             &mpt->m_phy_info[i].smhba_info);
7141         break;
7142     case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
7143         (void) sprintf(curr, "SATA OOB "
7144             "complete");
7145         break;
7146     case SMP_RESET_IN_PROGRESS:
7147         (void) sprintf(curr, "SMP reset in "
7148             "progress");
7149         break;
7150     case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
7151         (void) sprintf(curr, "is online at "
7152             "1.5 Gbps");
7153         if ((expd_handle == 0) &&
7154             (enc_handle == 1)) {

```

```

7155             mpt->m_port_chng = 1;
7156         }
7157         mptsas_smhba_log_sysevent(mpt,
7158             ESC_SAS_PHY_EVENT,
7159             SAS_PHY_ONLINE,
7160             &mpt->m_phy_info[i].smhba_info);
7161         break;
7162     case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
7163         (void) sprintf(curr, "is online at 3.0 "
7164             "Gbps");
7165         if ((expd_handle == 0) &&
7166             (enc_handle == 1)) {
7167             mpt->m_port_chng = 1;
7168         }
7169         mptsas_smhba_log_sysevent(mpt,
7170             ESC_SAS_PHY_EVENT,
7171             SAS_PHY_ONLINE,
7172             &mpt->m_phy_info[i].smhba_info);
7173         break;
7174     case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
7175         (void) sprintf(curr, "is online at "
7176             "6.0 Gbps");
7177         if ((expd_handle == 0) &&
7178             (enc_handle == 1)) {
7179             mpt->m_port_chng = 1;
7180         }
7181         mptsas_smhba_log_sysevent(mpt,
7182             ESC_SAS_PHY_EVENT,
7183             SAS_PHY_ONLINE,
7184             &mpt->m_phy_info[i].smhba_info);
7185         break;
7186     case MPI2_EVENT_SAS_TOPO_LR_RATE_12_0:
7187         (void) sprintf(curr, "is online at "
7188             "12.0 Gbps");
7189         if ((expd_handle == 0) &&
7190             (enc_handle == 1)) {
7191             mpt->m_port_chng = 1;
7192         }
7193         mptsas_smhba_log_sysevent(mpt,
7194             ESC_SAS_PHY_EVENT,
7195             SAS_PHY_ONLINE,
7196             &mpt->m_phy_info[i].smhba_info);
7197         break;
7198     default:
7199         (void) sprintf(curr, "state is "
7200             "unknown");
7201         break;
7202     }

7204     state = (link_rate &
7205         MPI2_EVENT_SAS_TOPO_LR_PREV_MASK) >>
7206         MPI2_EVENT_SAS_TOPO_LR_PREV_SHIFT;
7207     switch (state) {
7208     case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
7209         (void) sprintf(prev, "is disabled");
7210         break;
7211     case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
7212         (void) sprintf(prev, "was offline, "
7213             "failed speed negotiation");
7214         break;
7215     case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
7216         (void) sprintf(prev, "was SATA OOB "
7217             "complete");
7218         break;
7219     case SMP_RESET_IN_PROGRESS:
7220         (void) sprintf(prev, "was SMP reset "

```



```

7353         KM_SLEEP);
7354         topo_node->mpt = mpt;
7355         topo_node->un.phymask =
7356         ptgt->m_addr.mta_phymask;
7357         topo_node->event =
7358         MPTSAS_DR_EVENT_OFFLINE_TARGET;
7359         topo_node->devhdl = volhandle;
7360         topo_node->flags =
7361         MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
7362         topo_node->object = (void *)ptgt;
7363         if (topo_head == NULL) {
7364             topo_head = topo_tail = topo_node;
7365         } else {
7366             topo_tail->next = topo_node;
7367             topo_tail = topo_node;
7368         }
7369         break;
7370     }
7371     case MPI2_EVENT_IR_CHANGE_RC_PD_CREATED:
7372     case MPI2_EVENT_IR_CHANGE_RC_HIDE:
7373     {
7374         ptgt = rehash_linear_search(mpt->m_targets,
7375         mptsas_target_eval_devhdl, &diskhandle);
7376         if (ptgt == NULL)
7377             break;
7378
7379         /*
7380          * Update DR flag immediately avoid I/O failure
7381          */
7382         mutex_enter(&mpt->m_tx_waitq_mutex);
7383         ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
7384         mutex_exit(&mpt->m_tx_waitq_mutex);
7385
7386         topo_node = kmem_zalloc(
7387             sizeof (mptsas_topo_change_list_t),
7388             KM_SLEEP);
7389         topo_node->mpt = mpt;
7390         topo_node->un.phymask =
7391         ptgt->m_addr.mta_phymask;
7392         topo_node->event =
7393         MPTSAS_DR_EVENT_OFFLINE_TARGET;
7394         topo_node->devhdl = diskhandle;
7395         topo_node->flags =
7396         MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
7397         topo_node->object = (void *)ptgt;
7398         if (topo_head == NULL) {
7399             topo_head = topo_tail = topo_node;
7400         } else {
7401             topo_tail->next = topo_node;
7402             topo_tail = topo_node;
7403         }
7404         break;
7405     }
7406     case MPI2_EVENT_IR_CHANGE_RC_UNHIDE:
7407     case MPI2_EVENT_IR_CHANGE_RC_PD_DELETED:
7408     {
7409         /*
7410          * The physical drive is released by a IR
7411          * volume. But we cannot get the the physport
7412          * or phynum from the event data, so we only
7413          * can get the physport/phynum after SAS
7414          * Device Page0 request for the devhdl.
7415          */
7416         topo_node = kmem_zalloc(
7417             sizeof (mptsas_topo_change_list_t),
7418             KM_SLEEP);

```

```

7419         topo_node->mpt = mpt;
7420         topo_node->un.phymask = 0;
7421         topo_node->event =
7422         MPTSAS_DR_EVENT_RECONFIG_TARGET;
7423         topo_node->devhdl = diskhandle;
7424         topo_node->flags =
7425         MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
7426         topo_node->object = NULL;
7427         mpt->m_port_chng = 1;
7428         if (topo_head == NULL) {
7429             topo_head = topo_tail = topo_node;
7430         } else {
7431             topo_tail->next = topo_node;
7432             topo_tail = topo_node;
7433         }
7434         break;
7435     }
7436     default:
7437         break;
7438     }
7439 }
7440
7441 if (topo_head != NULL) {
7442     /*
7443      * Launch DR taskq to handle topology change
7444      */
7445     if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
7446         mptsas_handle_dr, (void *)topo_head,
7447         DDI_NOSLEEP)) != DDI_SUCCESS) {
7448         while (topo_head != NULL) {
7449             topo_node = topo_head;
7450             topo_head = topo_head->next;
7451             kmem_free(topo_node,
7452                 sizeof (mptsas_topo_change_list_t));
7453         }
7454         mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
7455             "for handle SAS DR event failed. \n");
7456     }
7457     break;
7458 }
7459 default:
7460     return (DDI_FAILURE);
7461 }
7462
7463 return (DDI_SUCCESS);
7464 }
7465 }
7466
7467 /*
7468  * handle events from ioc
7469  */
7470 static void
7471 mptsas_handle_event(void *args)
7472 {
7473     m_replyh_arg_t      *replyh_arg;
7474     pMpi2EventNotificationReply_t eventreply;
7475     uint32_t             event, iocloginfo, rfm;
7476     uint32_t             status;
7477     uint8_t              port;
7478     mptsas_t             *mpt;
7479     uint_t               iocstatus;
7480
7481     replyh_arg = (m_replyh_arg_t *)args;
7482     rfm = replyh_arg->rfm;
7483     mpt = replyh_arg->mpt;

```

```

7485 mutex_enter(&mpt->m_mutex);
7486 /*
7487  * If HBA is being reset, drop incoming event.
7488  */
7489 if (mpt->m_in_reset) {
7490     NDBG20(("dropping event received prior to reset"));
7491     mutex_exit(&mpt->m_mutex);
7492     return;
7493 }
7494
7495 eventreply = (pMpi2EventNotificationReply_t)
7496 (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
7497 event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);
7498
7499 if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
7500     &eventreply->IOCStatus)) {
7501     if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
7502         mptsas_log(mpt, CE_WARN,
7503             "!mptsas_handle_event: IOCStatus=0x%x, "
7504             "IOCLogInfo=0x%x", iocstatus,
7505             ddi_get32(mpt->m_acc_reply_frame_hdl,
7506                 &eventreply->IOCLogInfo));
7507     } else {
7508         mptsas_log(mpt, CE_WARN,
7509             "mptsas_handle_event: IOCStatus=0x%x, "
7510             "IOCLogInfo=0x%x", iocstatus,
7511             ddi_get32(mpt->m_acc_reply_frame_hdl,
7512                 &eventreply->IOCLogInfo));
7513     }
7514 }
7515
7516 /*
7517  * figure out what kind of event we got and handle accordingly
7518  */
7519 switch (event) {
7520 case MPI2_EVENT_LOG_ENTRY_ADDED:
7521     break;
7522 case MPI2_EVENT_LOG_DATA:
7523     iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7524         &eventreply->IOCLogInfo);
7525     NDBG20(("mptsas %d log info %x received.\n", mpt->m_instance,
7526         iocloginfo));
7527     break;
7528 case MPI2_EVENT_STATE_CHANGE:
7529     NDBG20(("mptsas%d state change.", mpt->m_instance));
7530     break;
7531 case MPI2_EVENT_HARD_RESET_RECEIVED:
7532     NDBG20(("mptsas%d event change.", mpt->m_instance));
7533     break;
7534 case MPI2_EVENT_SAS_DISCOVERY:
7535     {
7536         MPI2_EVENT_DATA_SAS_DISCOVERY *sasdiscovery;
7537         char string[80];
7538         uint8_t rc;
7539
7540         sasdiscovery =
7541             (pMpi2EventDataSasDiscovery_t)eventreply->EventData;
7542
7543         rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7544             &sasdiscovery->ReasonCode);
7545         port = ddi_get8(mpt->m_acc_reply_frame_hdl,
7546             &sasdiscovery->PhysicalPort);
7547         status = ddi_get32(mpt->m_acc_reply_frame_hdl,
7548             &sasdiscovery->DiscoveryStatus);
7549
7550         string[0] = 0;

```

```

7551     switch (rc) {
7552     case MPI2_EVENT_SAS_DISC_RC_STARTED:
7553         (void) sprintf(string, "STARTING");
7554         break;
7555     case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
7556         (void) sprintf(string, "COMPLETED");
7557         break;
7558     default:
7559         (void) sprintf(string, "UNKNOWN");
7560         break;
7561     }
7562
7563     NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
7564         port, status));
7565
7566     break;
7567 }
7568 case MPI2_EVENT_EVENT_CHANGE:
7569     NDBG20(("mptsas%d event change.", mpt->m_instance));
7570     break;
7571 case MPI2_EVENT_TASK_SET_FULL:
7572     {
7573         pMpi2EventDataTaskSetFull_t taskfull;
7574
7575         taskfull = (pMpi2EventDataTaskSetFull_t)eventreply->EventData;
7576
7577         NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
7578             mpt->m_instance, ddi_get16(mpt->m_acc_reply_frame_hdl,
7579                 &taskfull->CurrentDepth)));
7580         break;
7581     }
7582 case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
7583     {
7584         /*
7585          * SAS TOPOLOGY CHANGE LIST Event has already been handled
7586          * in mptsas_handle_event_sync() of interrupt context
7587          */
7588         break;
7589     }
7590 case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
7591     {
7592         pMpi2EventDataSasEnclDevStatusChange_t encstatus;
7593         uint8_t rc;
7594         char string[80];
7595
7596         encstatus = (pMpi2EventDataSasEnclDevStatusChange_t)
7597             eventreply->EventData;
7598
7599         rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7600             &encstatus->ReasonCode);
7601         switch (rc) {
7602         case MPI2_EVENT_SAS_ENCL_RC_ADDED:
7603             (void) sprintf(string, "added");
7604             break;
7605         case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
7606             (void) sprintf(string, ", not responding");
7607             break;
7608         default:
7609             break;
7610         }
7611         NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure "
7612             "%x%s\n", mpt->m_instance,
7613             ddi_get16(mpt->m_acc_reply_frame_hdl,
7614                 &encstatus->EnclosureHandle), string));
7615         break;
7616     }

```

```

7618  /*
7619  * MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by
7620  * mptsas_handle_event_sync,in here just send ack message.
7621  */
7622  case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
7623  {
7624      pMpi2EventDataSasDeviceStatusChange_t  statuschange;
7625      uint8_t                                  rc;
7626      uint16_t                                 devhdl;
7627      uint64_t                                 wwn = 0;
7628      uint32_t                                 wwn_lo, wwn_hi;

7630      statuschange = (pMpi2EventDataSasDeviceStatusChange_t)
7631      eventreply->EventData;
7632      rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7633      &statuschange->ReasonCode);
7634      wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7635      (uint32_t *) (void *)&statuschange->SASAddress);
7636      wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
7637      (uint32_t *) (void *)&statuschange->SASAddress + 1);
7638      wwn = ((uint64_t)wwn_hi << 32) | wwn_lo;
7639      devhdl = ddi_get16(mpt->m_acc_reply_frame_hdl,
7640      &statuschange->DevHandle);

7642      NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wwn is %"PRIx64,
7643      wwn));

7645      switch (rc) {
7646      case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
7647          NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
7648          ddi_get8(mpt->m_acc_reply_frame_hdl,
7649          &statuschange->ASC),
7650          ddi_get8(mpt->m_acc_reply_frame_hdl,
7651          &statuschange->ASCQ)));
7652          break;

7654      case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
7655          NDBG20(("Device not supported"));
7656          break;

7658      case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
7659          NDBG20(("IOC internally generated the Target Reset "
7660          "for devhdl:%x", devhdl));
7661          break;

7663      case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
7664          NDBG20(("IOC's internally generated Target Reset "
7665          "completed for devhdl:%x", devhdl));
7666          break;

7668      case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
7669          NDBG20(("IOC internally generated Abort Task"));
7670          break;

7672      case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
7673          NDBG20(("IOC's internally generated Abort Task "
7674          "completed"));
7675          break;

7677      case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
7678          NDBG20(("IOC internally generated Abort Task Set"));
7679          break;

7681      case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
7682          NDBG20(("IOC internally generated Clear Task Set"));

```

```

7683          break;

7685      case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
7686          NDBG20(("IOC internally generated Query Task"));
7687          break;

7689      case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
7690          NDBG20(("Device sent an Asynchronous Notification"));
7691          break;

7693      default:
7694          break;
7695      }
7696      break;
7697  }
7698  case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7699  {
7700      /*
7701      * IR TOPOLOGY CHANGE LIST Event has already been handled
7702      * in mpt_handle_event_sync() of interrupt context
7703      */
7704      break;
7705  }
7706  case MPI2_EVENT_IR_OPERATION_STATUS:
7707  {
7708      Mpi2EventDataIrOperationStatus_t      *irOpStatus;
7709      char                                    reason_str[80];
7710      uint8_t                                 rc, percent;
7711      uint16_t                                handle;

7713      irOpStatus = (pMpi2EventDataIrOperationStatus_t)
7714      eventreply->EventData;
7715      rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7716      &irOpStatus->RAIDOperation);
7717      percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
7718      &irOpStatus->PercentComplete);
7719      handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7720      &irOpStatus->VolDevHandle);

7722      switch (rc) {
7723      case MPI2_EVENT_IR_RAIDOP_RESYNC:
7724          (void) sprintf(reason_str, "resync");
7725          break;
7726      case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
7727          (void) sprintf(reason_str, "online capacity "
7728          "expansion");
7729          break;
7730      case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
7731          (void) sprintf(reason_str, "consistency check");
7732          break;
7733      default:
7734          (void) sprintf(reason_str, "unknown reason %x",
7735          rc);
7736      }

7738      NDBG20(("mptsas%d raid operational status: (%s)"
7739      "\thandle(0x%04x), percent complete(%d)\n",
7740      mpt->m_instance, reason_str, handle, percent));
7741      break;
7742  }
7743  case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
7744  {
7745      pMpi2EventDataSasBroadcastPrimitive_t  sas_broadcast;
7746      uint8_t                                  phy_num;
7747      uint8_t                                  primitive;

```

```

7749     sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
7750     eventreply->EventData;

7752     phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
7753     &sas_broadcast->PhyNum);
7754     primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
7755     &sas_broadcast->Primitive);

7757     switch (primitive) {
7758     case MPI2_EVENT_PRIMITIVE_CHANGE:
7759         mptsas_smhba_log_sysevent(mpt,
7760         ESC_SAS_HBA_PORT_BROADCAST,
7761         SAS_PORT_BROADCAST_CHANGE,
7762         &mpt->m_phy_info[phy_num].smhba_info);
7763         break;
7764     case MPI2_EVENT_PRIMITIVE_SES:
7765         mptsas_smhba_log_sysevent(mpt,
7766         ESC_SAS_HBA_PORT_BROADCAST,
7767         SAS_PORT_BROADCAST_SES,
7768         &mpt->m_phy_info[phy_num].smhba_info);
7769         break;
7770     case MPI2_EVENT_PRIMITIVE_EXPANDER:
7771         mptsas_smhba_log_sysevent(mpt,
7772         ESC_SAS_HBA_PORT_BROADCAST,
7773         SAS_PORT_BROADCAST_D01_4,
7774         &mpt->m_phy_info[phy_num].smhba_info);
7775         break;
7776     case MPI2_EVENT_PRIMITIVE_ASYNCHRONOUS_EVENT:
7777         mptsas_smhba_log_sysevent(mpt,
7778         ESC_SAS_HBA_PORT_BROADCAST,
7779         SAS_PORT_BROADCAST_D04_7,
7780         &mpt->m_phy_info[phy_num].smhba_info);
7781         break;
7782     case MPI2_EVENT_PRIMITIVE_RESERVED3:
7783         mptsas_smhba_log_sysevent(mpt,
7784         ESC_SAS_HBA_PORT_BROADCAST,
7785         SAS_PORT_BROADCAST_D16_7,
7786         &mpt->m_phy_info[phy_num].smhba_info);
7787         break;
7788     case MPI2_EVENT_PRIMITIVE_RESERVED4:
7789         mptsas_smhba_log_sysevent(mpt,
7790         ESC_SAS_HBA_PORT_BROADCAST,
7791         SAS_PORT_BROADCAST_D29_7,
7792         &mpt->m_phy_info[phy_num].smhba_info);
7793         break;
7794     case MPI2_EVENT_PRIMITIVE_CHANGE0_RESERVED:
7795         mptsas_smhba_log_sysevent(mpt,
7796         ESC_SAS_HBA_PORT_BROADCAST,
7797         SAS_PORT_BROADCAST_D24_0,
7798         &mpt->m_phy_info[phy_num].smhba_info);
7799         break;
7800     case MPI2_EVENT_PRIMITIVE_CHANGE1_RESERVED:
7801         mptsas_smhba_log_sysevent(mpt,
7802         ESC_SAS_HBA_PORT_BROADCAST,
7803         SAS_PORT_BROADCAST_D27_4,
7804         &mpt->m_phy_info[phy_num].smhba_info);
7805         break;
7806     default:
7807         NDBG16(("mptsas%d: unknown BROADCAST PRIMITIVE"
7808         " %x received",
7809         mpt->m_instance, primitive));
7810         break;
7811     }
7812     NDBG16(("mptsas%d sas broadcast primitive: "
7813     "\tprimitive(0x%04x), phy(%d) complete\n",
7814     mpt->m_instance, primitive, phy_num));

```

```

7815         break;
7816     }
7817     case MPI2_EVENT_IR_VOLUME:
7818     {
7819         Mpi2EventDataIrVolume_t      *irVolume;
7820         uint16_t                      devhandle;
7821         uint32_t                      state;
7822         int                            config, vol;
7823         uint8_t                        found = FALSE;

7825         irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
7826         state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7827         &irVolume->NewValue);
7828         devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7829         &irVolume->VolDevHandle);

7831         NDBG20(("EVENT_IR_VOLUME event is received"));

7833         /*
7834         * Get latest RAID info and then find the DevHandle for this
7835         * event in the configuration. If the DevHandle is not found
7836         * just exit the event.
7837         */
7838         (void) mptsas_get_raid_info(mpt);
7839         for (config = 0; (config < mpt->m_num_raid_configs) &&
7840         (!found); config++) {
7841             for (vol = 0; vol < MPTSAS_MAX_RAIDVOL; vol++) {
7842                 if (mpt->m_raidconfig[config].m_raidvol[vol].
7843                 m_raidhandle == devhandle) {
7844                     found = TRUE;
7845                     break;
7846                 }
7847             }
7848         }
7849         if (!found) {
7850             break;
7851         }

7853         switch (irVolume->ReasonCode) {
7854         case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
7855         {
7856             uint32_t i;
7857             mpt->m_raidconfig[config].m_raidvol[vol].m_settings =
7858             state;

7860             i = state & MPI2_RAIDVOL0_SETTING_MASK_WRITE_CACHING;
7861             mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
7862             ", auto-config of hot-swap drives is %s"
7863             ", write caching is %s"
7864             ", hot-spare pool mask is %02x\n",
7865             vol, state &
7866             MPI2_RAIDVOL0_SETTING_AUTO_CONFIG_HSWAP_DISABLE
7867             ? "disabled" : "enabled",
7868             i == MPI2_RAIDVOL0_SETTING_UNCHANGED
7869             ? "controlled by member disks" :
7870             i == MPI2_RAIDVOL0_SETTING_DISABLE_WRITE_CACHING
7871             ? "disabled" :
7872             i == MPI2_RAIDVOL0_SETTING_ENABLE_WRITE_CACHING
7873             ? "enabled" :
7874             "incorrectly set",
7875             (state >> 16) & 0xff);
7876             break;
7877         }
7878         case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
7879         {
7880             mpt->m_raidconfig[config].m_raidvol[vol].m_state =

```

```

7881         (uint8_t)state;
7882
7883         mptsas_log(mpt, CE_NOTE,
7884                 "Volume %d is now %s\n", vol,
7885                 state == MPI2_RAID_VOL_STATE_OPTIMAL
7886                 ? "optimal" :
7887                 state == MPI2_RAID_VOL_STATE_DEGRADED
7888                 ? "degraded" :
7889                 state == MPI2_RAID_VOL_STATE_ONLINE
7890                 ? "online" :
7891                 state == MPI2_RAID_VOL_STATE_INITIALIZING
7892                 ? "initializing" :
7893                 state == MPI2_RAID_VOL_STATE_FAILED
7894                 ? "failed" :
7895                 state == MPI2_RAID_VOL_STATE_MISSING
7896                 ? "missing" :
7897                 "state unknown");
7898         break;
7899     }
7900     case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
7901     {
7902         mpt->m_raidconfig[config].m_raidvol[vol].
7903             m_statusflags = state;
7904
7905         mptsas_log(mpt, CE_NOTE,
7906                 " Volume %d is now %s%s%s%s%s%s\n",
7907                 vol,
7908                 state & MPI2_RAIDVOL0_STATUS_FLAG_ENABLED
7909                 ? ", enabled" : ", disabled",
7910                 state & MPI2_RAIDVOL0_STATUS_FLAG_QUIESCED
7911                 ? ", quiesced" : "",
7912                 state & MPI2_RAIDVOL0_STATUS_FLAG_VOLUME_INACTIVE
7913                 ? ", inactive" : ", active",
7914                 state &
7915                 MPI2_RAIDVOL0_STATUS_FLAG_BAD_BLOCK_TABLE_FULL
7916                 ? ", bad block table is full" : "",
7917                 state &
7918                 MPI2_RAIDVOL0_STATUS_FLAG_RESYNC_IN_PROGRESS
7919                 ? ", resync in progress" : "",
7920                 state & MPI2_RAIDVOL0_STATUS_FLAG_BACKGROUND_INIT
7921                 ? ", background initialization in progress" : "",
7922                 state &
7923                 MPI2_RAIDVOL0_STATUS_FLAG_CAPACITY_EXPANSION
7924                 ? ", capacity expansion in progress" : "",
7925                 state &
7926                 MPI2_RAIDVOL0_STATUS_FLAG_CONSISTENCY_CHECK
7927                 ? ", consistency check in progress" : "",
7928                 state & MPI2_RAIDVOL0_STATUS_FLAG_DATA_SCRUB
7929                 ? ", data scrub in progress" : "");
7930         break;
7931     }
7932     default:
7933         break;
7934     }
7935     break;
7936 }
7937 case MPI2_EVENT_IR_PHYSICAL_DISK:
7938 {
7939     Mpi2EventDataIrPhysicalDisk_t *irPhysDisk;
7940     uint16_t devhandle, enchandle, slot;
7941     uint32_t status, state;
7942     uint8_t physdisknum, reason;
7943
7944     irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
7945         eventreply->EventData;
7946     physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,

```

```

7947         &irPhysDisk->PhysDiskNum);
7948     devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7949         &irPhysDisk->PhysDiskDevHandle);
7950     enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7951         &irPhysDisk->EnclosureHandle);
7952     slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
7953         &irPhysDisk->Slot);
7954     state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7955         &irPhysDisk->NewValue);
7956     reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
7957         &irPhysDisk->ReasonCode);
7958
7959     NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));
7960
7961     switch (reason) {
7962     case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
7963         mptsas_log(mpt, CE_NOTE,
7964                 " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7965                 "for enclosure with handle 0x%x is now in hot "
7966                 "spare pool %d",
7967                 physdisknum, devhandle, slot, enchandle,
7968                 (state >> 16) & 0xff);
7969         break;
7970
7971     case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
7972         status = state;
7973         mptsas_log(mpt, CE_NOTE,
7974                 " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7975                 "for enclosure with handle 0x%x is now "
7976                 "%s%s%s%s\n", physdisknum, devhandle, slot,
7977                 enchandle,
7978                 status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
7979                 ? ", inactive" : ", active",
7980                 status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
7981                 ? ", out of sync" : "",
7982                 status & MPI2_PHYSDISK0_STATUS_FLAG_QUIESCED
7983                 ? ", quiesced" : "",
7984                 status &
7985                 MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
7986                 ? ", write cache enabled" : "",
7987                 status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
7988                 ? ", capacity expansion target" : "");
7989         break;
7990
7991     case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
7992         mptsas_log(mpt, CE_NOTE,
7993                 " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7994                 "for enclosure with handle 0x%x is now %s\n",
7995                 physdisknum, devhandle, slot, enchandle,
7996                 state == MPI2_RAID_PD_STATE_OPTIMAL
7997                 ? "optimal" :
7998                 state == MPI2_RAID_PD_STATE_REBUILDING
7999                 ? "rebuilding" :
8000                 state == MPI2_RAID_PD_STATE_DEGRADED
8001                 ? "degraded" :
8002                 state == MPI2_RAID_PD_STATE_HOT_SPARE
8003                 ? "a hot spare" :
8004                 state == MPI2_RAID_PD_STATE_ONLINE
8005                 ? "online" :
8006                 state == MPI2_RAID_PD_STATE_OFFLINE
8007                 ? "offline" :
8008                 state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
8009                 ? "not compatible" :
8010                 state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
8011                 ? "not configured" :
8012                 "state unknown");

```

```

8013         break;
8014     }
8015     break;
8016 }
8017 default:
8018     NDBG20(("mptsas%d: unknown event %x received",
8019         mpt->m_instance, event));
8020     break;
8021 }
8022
8023 /*
8024  * Return the reply frame to the free queue.
8025  */
8026 ddi_put32(mpt->m_acc_free_queue_hdl,
8027     &(uint32_t *) (void *) mpt->m_free_queue[mpt->m_free_index], rfm);
8028 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
8029     DDI_DMA_SYNC_FORDEV);
8030 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
8031     mpt->m_free_index = 0;
8032 }
8033 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
8034     mpt->m_free_index);
8035 mutex_exit(&mpt->m_mutex);
8036 }
8037
8038 /*
8039  * invoked from timeout() to restart qfull cmds with throttle == 0
8040  */
8041 static void
8042 mptsas_restart_cmd(void *arg)
8043 {
8044     mptsas_t *mpt = arg;
8045     mptsas_target_t *ptgt = NULL;
8046
8047     mutex_enter(&mpt->m_mutex);
8048
8049     mpt->m_restart_cmd_timeid = 0;
8050
8051     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
8052         ptgt = rehash_next(mpt->m_targets, ptgt)) {
8053         if (ptgt->m_reset_delay == 0) {
8054             if (ptgt->m_t_throttle == QFULL_THROTTLE) {
8055                 mptsas_set_throttle(mpt, ptgt,
8056                     MAX_THROTTLE);
8057             }
8058         }
8059     }
8060     mptsas_restart_hba(mpt);
8061     mutex_exit(&mpt->m_mutex);
8062 }
8063
8064 void
8065 mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
8066 {
8067     int slot;
8068     mptsas_slots_t *slots = mpt->m_active;
8069     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8070
8071     ASSERT(cmd != NULL);
8072     ASSERT(cmd->cmd_queued == FALSE);
8073
8074     /*
8075      * Task Management cmds are removed in their own routines. Also,
8076      * we don't want to modify timeout based on TM cmds.
8077      */
8078     if (cmd->cmd_flags & CFLAG_TM_CMD) {

```

```

8079         return;
8080     }
8081
8082     slot = cmd->cmd_slot;
8083
8084     /*
8085      * remove the cmd.
8086      */
8087     if (cmd == slots->m_slot[slot]) {
8088         NDBG31(("mptsas_remove_cmd: removing cmd=0x%p, flags "
8089             "0x%x", (void *) cmd, cmd->cmd_flags));
8090         slots->m_slot[slot] = NULL;
8091         mpt->m_ncmds--;
8092
8093         /*
8094          * only decrement per target ncmds if command
8095          * has a target associated with it.
8096          */
8097         if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
8098             ptgt->m_t_ncmds--;
8099             /*
8100              * reset throttle if we just ran an untagged command
8101              * to a tagged target
8102              */
8103             if ((ptgt->m_t_ncmds == 0) &&
8104                 ((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0)) {
8105                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
8106             }
8107
8108             /*
8109              * Remove this command from the active queue.
8110              */
8111             if (cmd->cmd_active_expiration != 0) {
8112                 TAILQ_REMOVE(&ptgt->m_active_cmdq, cmd,
8113                     cmd_active_link);
8114                 cmd->cmd_active_expiration = 0;
8115             }
8116         }
8117     }
8118
8119     /*
8120      * This is all we need to do for ioc commands.
8121      */
8122     if (cmd->cmd_flags & CFLAG_CMDIOC) {
8123         mptsas_return_to_pool(mpt, cmd);
8124         return;
8125     }
8126
8127     ASSERT(cmd != slots->m_slot[cmd->cmd_slot]);
8128 }
8129
8130 /*
8131  * accept all cmds on the tx_waitq if any and then
8132  * start a fresh request from the top of the device queue.
8133  */
8134 * since there are always cmds queued on the tx_waitq, and rare cmds on
8135 * the instance waitq, so this function should not be invoked in the ISR,
8136 * the mptsas_restart_waitq() is invoked in the ISR instead. otherwise, the
8137 * burden belongs to the IO dispatch CPUs is moved the interrupt CPU.
8138 */
8139 static void
8140 mptsas_restart_hba(mptsas_t *mpt)
8141 {
8142     ASSERT(mutex_owned(&mpt->m_mutex));
8143
8144     mutex_enter(&mpt->m_tx_waitq_mutex);

```

```

8145     if (mpt->m_tx_waitq) {
8146         mptsas_accept_tx_waitq(mpt);
8147     }
8148     mutex_exit(&mpt->m_tx_waitq_mutex);
8149     mptsas_restart_waitq(mpt);
8150 }

8152 /*
8153  * start a fresh request from the top of the device queue
8154  */
8155 static void
8156 mptsas_restart_waitq(mptsas_t *mpt)
8157 {
8158     mptsas_cmd_t *cmd, *next_cmd;
8159     mptsas_target_t *tgt = NULL;

8161     NDBG1(("mptsas_restart_waitq: mpt=0x%p", (void *)mpt));

8163     ASSERT(mutex_owned(&mpt->m_mutex));

8165     /*
8166      * If there is a reset delay, don't start any cmds.  Otherwise, start
8167      * as many cmds as possible.
8168      * Since SMID 0 is reserved and the TM slot is reserved, the actual max
8169      * commands is m_max_requests - 2.
8170      */
8171     cmd = mpt->m_waitq;

8173     while (cmd != NULL) {
8174         next_cmd = cmd->cmd_linkp;
8175         if (cmd->cmd_flags & CFLAG_PASSTHRU) {
8176             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8177                 /*
8178                  * passthru command get slot need
8179                  * set CFLAG_PREPARED.
8180                  */
8181                 cmd->cmd_flags |= CFLAG_PREPARED;
8182                 mptsas_waitq_delete(mpt, cmd);
8183                 mptsas_start_passthru(mpt, cmd);
8184             }
8185             cmd = next_cmd;
8186             continue;
8187         }
8188         if (cmd->cmd_flags & CFLAG_CONFIG) {
8189             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8190                 /*
8191                  * Send the config page request and delete it
8192                  * from the waitq.
8193                  */
8194                 cmd->cmd_flags |= CFLAG_PREPARED;
8195                 mptsas_waitq_delete(mpt, cmd);
8196                 mptsas_start_config_page_access(mpt, cmd);
8197             }
8198             cmd = next_cmd;
8199             continue;
8200         }
8201         if (cmd->cmd_flags & CFLAG_FW_DIAG) {
8202             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8203                 /*
8204                  * Send the FW Diag request and delete if from
8205                  * the waitq.
8206                  */
8207                 cmd->cmd_flags |= CFLAG_PREPARED;
8208                 mptsas_waitq_delete(mpt, cmd);
8209                 mptsas_start_diag(mpt, cmd);
8210             }

```

```

8211         cmd = next_cmd;
8212         continue;
8213     }

8215     tgt = cmd->cmd_tgt_addr;
8216     if (tgt && (tgt->m_t_throttle == DRAIN_THROTTLE) &&
8217         (tgt->m_t_ncmds == 0)) {
8218         mptsas_set_throttle(mpt, tgt, MAX_THROTTLE);
8219     }
8220     if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
8221         (tgt && (tgt->m_reset_delay == 0)) &&
8222         (tgt && (tgt->m_t_ncmds <
8223             (tgt->m_t_throttle)))) {
8224         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8225             mptsas_waitq_delete(mpt, cmd);
8226             (void) mptsas_start_cmd(mpt, cmd);
8227         }
8228     }
8229     cmd = next_cmd;
8230 }
8231 }
8232 /*
8233  * Cmds are queued if tran_start() doesn't get the m_mutexlock(no wait).
8234  * Accept all those queued cmds before new cmd is accept so that the
8235  * cmds are sent in order.
8236  */
8237 static void
8238 mptsas_accept_tx_waitq(mptsas_t *mpt)
8239 {
8240     mptsas_cmd_t *cmd;

8242     ASSERT(mutex_owned(&mpt->m_mutex));
8243     ASSERT(mutex_owned(&mpt->m_tx_waitq_mutex));

8245     /*
8246      * A Bus Reset could occur at any time and flush the tx_waitq,
8247      * so we cannot count on the tx_waitq to contain even one cmd.
8248      * And when the m_tx_waitq_mutex is released and run
8249      * mptsas_accept_pkt(), the tx_waitq may be flushed.
8250      */
8251     cmd = mpt->m_tx_waitq;
8252     for (;;) {
8253         if ((cmd = mpt->m_tx_waitq) == NULL) {
8254             mpt->m_tx_draining = 0;
8255             break;
8256         }
8257         if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL) {
8258             mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
8259         }
8260         cmd->cmd_linkp = NULL;
8261         mutex_exit(&mpt->m_tx_waitq_mutex);
8262         if (mptsas_accept_pkt(mpt, cmd) != TRAN_ACCEPT)
8263             cmn_err(CE_WARN, "mpt: mptsas_accept_tx_waitq: failed "
8264                 "to accept cmd on queue\n");
8265         mutex_enter(&mpt->m_tx_waitq_mutex);
8266     }
8267 }

8270 /*
8271  * mpt tag type lookup
8272  */
8273 static char mptsas_tag_lookup[] =
8274     {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};

8276 static int

```

```

8277 mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
8278 {
8279     struct scsi_pkt      *pkt = CMD2PKT(cmd);
8280     uint32_t             control = 0;
8281     caddr_t              mem;
8282     pMpi2SCSIIORequest_t io_request;
8283     ddi_dma_handle_t     dma_hdl = mpt->m_dma_req_frame_hdl;
8284     ddi_acc_handle_t     acc_hdl = mpt->m_acc_req_frame_hdl;
8285     mptsas_target_t      *ptgt = cmd->cmd_tgt_addr;
8286     uint16_t             SMID, io_flags = 0;
8287     uint32_t             request_desc_low, request_desc_high;
8288     mptsas_cmd_t         *c;

8290     NDBG1(("mptsas_start_cmd: cmd=0x%p, flags 0x%x", (void *)cmd,
8291           cmd->cmd_flags));

8293     /*
8294      * Set SMID and increment index. Rollover to 1 instead of 0 if index
8295      * is at the max. 0 is an invalid SMID, so we call the first index 1.
8296      */
8297     SMID = cmd->cmd_slot;

8299     /*
8300      * It is possible for back to back device reset to
8301      * happen before the reset delay has expired. That's
8302      * ok, just let the device reset go out on the bus.
8303      */
8304     if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
8305         ASSERT(ptgt->m_reset_delay == 0);
8306     }

8308     /*
8309      * if a non-tagged cmd is submitted to an active tagged target
8310      * then drain before submitting this cmd; SCSI-2 allows RQSENSE
8311      * to be untagged
8312      */
8313     if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
8314         (ptgt->m_t_ncmds > 1) &&
8315         ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
8316         (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
8317         if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
8318             NDBG23(("target=%d, untagged cmd, start draining\n",
8319                   ptgt->m_devhdl));

8321             if (ptgt->m_reset_delay == 0) {
8322                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
8323             }

8325             mptsas_remove_cmd(mpt, cmd);
8326             cmd->cmd_pkt_flags |= FLAG_HEAD;
8327             mptsas_waitq_add(mpt, cmd);
8328         }
8329         return (DDI_FAILURE);
8330     }

8332     /*
8333      * Set correct tag bits.
8334      */
8335     if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
8336         switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
8337             FLAG_TAGMASK) >> 12)]) {
8338             case MSG_SIMPLE_QTAG:
8339                 control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
8340                 break;
8341             case MSG_HEAD_QTAG:
8342                 control |= MPI2_SCSIIO_CONTROL_HEADOFQ;

```

```

8343         break;
8344         case MSG_ORDERED_QTAG:
8345             control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
8346             break;
8347         default:
8348             mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
8349             break;
8350     }
8351 } else {
8352     if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
8353         ptgt->m_t_throttle = 1;
8354     }
8355     control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
8356 }

8358 if (cmd->cmd_pkt_flags & FLAG_TLR) {
8359     control |= MPI2_SCSIIO_CONTROL_TLR_ON;
8360 }

8362 mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
8363 io_request = (pMpi2SCSIIORequest_t)mem;

8365 bzero(io_request, sizeof (Mpi2SCSIIORequest_t));
8366 ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
8367         (MPI2_SCSI_IO_REQUEST, SGL) / 4);
8368 mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
8369         MPI2_FUNCTION_SCSI_IO_REQUEST);

8371 (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
8372                   io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);

8374 io_flags = cmd->cmd_cdblen;
8375 if (mptsas_use_fastpath &&
8376     ptgt->m_io_flags & MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
8377     io_flags |= MPI25_SCSIIO_IOFLAGS_FAST_PATH;
8378     request_desc_low = MPI25_REQ_DESCRIPTOR_FLAGS_FAST_PATH_SCSI_IO;
8379 } else {
8380     request_desc_low = MPI2_REQ_DESCRIPTOR_FLAGS_SCSI_IO;
8381 }
8382 ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
8383 /*
8384  * setup the Scatter/Gather DMA list for this request
8385  */
8386 if (cmd->cmd_cookiec > 0) {
8387     mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
8388 } else {
8389     ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
8390             ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
8391             MPI2_SGE_FLAGS_END_OF_BUFFER |
8392             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
8393             MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
8394 }

8396 /*
8397  * save ARQ information
8398  */
8399 ddi_put8(acc_hdl, &io_request->SenseBufferLength, cmd->cmd_rqslen);
8400 if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUNVALID)) ==
8401     (CFLAG_SCBEXTERN | CFLAG_EXTARQBUNVALID)) {
8402     ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
8403             cmd->cmd_ext_arqcookie.dmac_address);
8404 } else {
8405     ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
8406             cmd->cmd_arqcookie.dmac_address);
8407 }

```



```

8409     ddi_put32(acc_hdl, &io_request->Control, control);

8411     NDBG31(("starting message=%d(0x%p), with cmd=0x%p",
8412           SMID, (void *)io_request, (void *)cmd));

8414     (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);

8416     /*
8417     * Build request descriptor and write it to the request desc post reg.
8418     */
8419     request_desc_low |= (SMID << 16);
8420     request_desc_high = ptgt->m_devhdl << 16;
8421     MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);

8423     /*
8424     * Start timeout.
8425     */
8426     cmd->cmd_active_expiration =
8427         gethrtime() + (hrtime_t)pkt->pkt_time * NANOSEC;
8428 #ifdef MPTSAS_TEST
8429     /*
8430     * Force timeouts to happen immediately.
8431     */
8432     if (mptsas_test_timeouts)
8433         cmd->cmd_active_expiration = gethrtime();
8434 #endif
8435     c = TAILQ_FIRST(&ptgt->m_active_cmdq);
8436     if (c == NULL ||
8437         c->cmd_active_expiration < cmd->cmd_active_expiration) {
8438         /*
8439         * Common case is that this is the last pending expiration
8440         * (or queue is empty). Insert at head of the queue.
8441         */
8442         TAILQ_INSERT_HEAD(&ptgt->m_active_cmdq, cmd, cmd_active_link);
8443     } else {
8444         /*
8445         * Queue is not empty and first element expires later than
8446         * this command. Search for element expiring sooner.
8447         */
8448         while ((c = TAILQ_NEXT(c, cmd_active_link)) != NULL) {
8449             if (c->cmd_active_expiration <
8450                 cmd->cmd_active_expiration) {
8451                 TAILQ_INSERT_BEFORE(c, cmd, cmd_active_link);
8452                 break;
8453             }
8454         }
8455         if (c == NULL) {
8456             /*
8457             * No element found expiring sooner, append to
8458             * non-empty queue.
8459             */
8460             TAILQ_INSERT_TAIL(&ptgt->m_active_cmdq, cmd,
8461                             cmd_active_link);
8462         }
8463     }

8465     if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
8466         (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
8467         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8468         return (DDI_FAILURE);
8469     }
8470     return (DDI_SUCCESS);
8471 }

8473 /*
8474 * Select a helper thread to handle current doneq

```

```

8475     /*
8476     static void
8477     mptsas_deliver_doneq_thread(mptsas_t *mpt)
8478     {
8479         uint64_t            t, i;
8480         uint32_t            min = 0xffffffff;
8481         mptsas_doneq_thread_list_t *item;

8483         for (i = 0; i < mpt->m_doneq_thread_n; i++) {
8484             item = &mpt->m_doneq_thread_id[i];
8485             /*
8486             * If the completed command on help thread[i] less than
8487             * doneq_thread_threshold, then pick the thread[i]. Otherwise
8488             * pick a thread which has least completed command.
8489             */

8491             mutex_enter(&item->mutex);
8492             if (item->len < mpt->m_doneq_thread_threshold) {
8493                 t = i;
8494                 mutex_exit(&item->mutex);
8495                 break;
8496             }
8497             if (item->len < min) {
8498                 min = item->len;
8499                 t = i;
8500             }
8501             mutex_exit(&item->mutex);
8502         }
8503         mutex_enter(&mpt->m_doneq_thread_id[t].mutex);
8504         mptsas_doneq_mv(mpt, t);
8505         cv_signal(&mpt->m_doneq_thread_id[t].cv);
8506         mutex_exit(&mpt->m_doneq_thread_id[t].mutex);
8507     }

8509     /*
8510     * move the current global doneq to the doneq of thead[t]
8511     */
8512     static void
8513     mptsas_doneq_mv(mptsas_t *mpt, uint64_t t)
8514     {
8515         mptsas_cmd_t            *cmd;
8516         mptsas_doneq_thread_list_t *item = &mpt->m_doneq_thread_id[t];

8518         ASSERT(mutex_owned(&item->mutex));
8519         while ((cmd = mpt->m_doneq) != NULL) {
8520             if ((mpt->m_doneq = cmd->cmd_linkp) == NULL) {
8521                 mpt->m_donetail = &mpt->m_doneq;
8522             }
8523             cmd->cmd_linkp = NULL;
8524             *item->donetail = cmd;
8525             item->donetail = &cmd->cmd_linkp;
8526             mpt->m_doneq_len--;
8527             item->len++;
8528         }
8529     }

8531     void
8532     mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd)
8533     {
8534         struct scsi_pkt *pkt = CMD2PKT(cmd);

8536         /* Check all acc and dma handles */
8537         if ((mptsas_check_acc_handle(mpt->m_datap) !=
8538             DDI_SUCCESS) ||
8539             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
8540             DDI_SUCCESS) ||

```

```

8541     (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
8542     DDI_SUCCESS) ||
8543     (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
8544     DDI_SUCCESS) ||
8545     (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
8546     DDI_SUCCESS) ||
8547     (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
8548     DDI_SUCCESS) ||
8549     (mptsas_check_acc_handle(mpt->m_config_handle) !=
8550     DDI_SUCCESS)) {
8551         ddi_fm_service_impact(mpt->m_dip,
8552         DDI_SERVICE_UNAFFECTED);
8553         ddi_fm_acc_err_clear(mpt->m_config_handle,
8554         DDI_FME_VER0);
8555         pkt->pkt_reason = CMD_TRAN_ERR;
8556         pkt->pkt_statistics = 0;
8557     }
8558     if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
8559     DDI_SUCCESS) ||
8560     (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
8561     DDI_SUCCESS) ||
8562     (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
8563     DDI_SUCCESS) ||
8564     (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
8565     DDI_SUCCESS) ||
8566     (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=
8567     DDI_SUCCESS)) {
8568         ddi_fm_service_impact(mpt->m_dip,
8569         DDI_SERVICE_UNAFFECTED);
8570         pkt->pkt_reason = CMD_TRAN_ERR;
8571         pkt->pkt_statistics = 0;
8572     }
8573     if (cmd->cmd_dmahandle &&
8574     (mptsas_check_dma_handle(cmd->cmd_dmahandle) != DDI_SUCCESS)) {
8575         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8576         pkt->pkt_reason = CMD_TRAN_ERR;
8577         pkt->pkt_statistics = 0;
8578     }
8579     if ((cmd->cmd_extra_frames &&
8580     ((mptsas_check_dma_handle(cmd->cmd_extra_frames->m_dma_hdl) !=
8581     DDI_SUCCESS) ||
8582     (mptsas_check_acc_handle(cmd->cmd_extra_frames->m_acc_hdl) !=
8583     DDI_SUCCESS)))) {
8584         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8585         pkt->pkt_reason = CMD_TRAN_ERR;
8586         pkt->pkt_statistics = 0;
8587     }
8588     if (cmd->cmd_arqhandle &&
8589     (mptsas_check_dma_handle(cmd->cmd_arqhandle) != DDI_SUCCESS)) {
8590         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8591         pkt->pkt_reason = CMD_TRAN_ERR;
8592         pkt->pkt_statistics = 0;
8593     }
8594     if (cmd->cmd_ext_arqhandle &&
8595     (mptsas_check_dma_handle(cmd->cmd_ext_arqhandle) != DDI_SUCCESS)) {
8596         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8597         pkt->pkt_reason = CMD_TRAN_ERR;
8598         pkt->pkt_statistics = 0;
8599     }
8600 }

8602 /*
8603  * These routines manipulate the queue of commands that
8604  * are waiting for their completion routines to be called.
8605  * The queue is usually in FIFO order but on an MP system
8606  * it's possible for the completion routines to get out

```

```

8607  * of order. If that's a problem you need to add a global
8608  * mutex around the code that calls the completion routine
8609  * in the interrupt handler.
8610  */
8611 static void
8612 mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8613 {
8614     struct scsi_pkt *pkt = CMD2PKT(cmd);

8616     NDBG31(("mptsas_doneq add: cmd=0x%p", (void *)cmd));

8618     ASSERT((cmd->cmd_flags & CFLAG_COMPLETED) == 0);
8619     cmd->cmd_linkp = NULL;
8620     cmd->cmd_flags |= CFLAG_FINISHED;
8621     cmd->cmd_flags &= ~CFLAG_IN_TRANSPORT;

8623     mptsas_fma_check(mpt, cmd);

8625     /*
8626     * only add scsi pkts that have completion routines to
8627     * the doneq. no intr cmds do not have callbacks.
8628     */
8629     if (pkt && (pkt->pkt_comp)) {
8630         *mpt->m_donetail = cmd;
8631         mpt->m_donetail = &cmd->cmd_linkp;
8632         mpt->m_doneq_len++;
8633     }
8634 }

8636 static mptsas_cmd_t *
8637 mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t)
8638 {
8639     mptsas_cmd_t *cmd;
8640     mptsas_doneq_thread_list_t *item = &mpt->m_doneq_thread_id[t];

8642     /* pop one off the done queue */
8643     if ((cmd = item->doneq) != NULL) {
8644         /* if the queue is now empty fix the tail pointer */
8645         NDBG31(("mptsas_doneq_thread_rm: cmd=0x%p", (void *)cmd));
8646         if ((item->doneq = cmd->cmd_linkp) == NULL) {
8647             item->donetail = &item->doneq;
8648         }
8649         cmd->cmd_linkp = NULL;
8650         item->len--;
8651     }
8652     return (cmd);
8653 }

8655 static void
8656 mptsas_doneq_empty(mptsas_t *mpt)
8657 {
8658     if (mpt->m_doneq && !mpt->m_in_callback) {
8659         mptsas_cmd_t *cmd, *next;
8660         struct scsi_pkt *pkt;

8662         mpt->m_in_callback = 1;
8663         cmd = mpt->m_doneq;
8664         mpt->m_doneq = NULL;
8665         mpt->m_donetail = &mpt->m_doneq;
8666         mpt->m_doneq_len = 0;

8668         mutex_exit(&mpt->m_mutex);
8669         /*
8670         * run the completion routines of all the
8671         * completed commands
8672         */

```

```

8673     while (cmd != NULL) {
8674         next = cmd->cmd_linkp;
8675         cmd->cmd_linkp = NULL;
8676         /* run this command's completion routine */
8677         cmd->cmd_flags |= CFLAG_COMPLETED;
8678         pkt = CMD2PKT(cmd);
8679         mptsas_pkt_comp(pkt, cmd);
8680         cmd = next;
8681     }
8682     mutex_enter(&mpt->m_mutex);
8683     mpt->m_in_callback = 0;
8684 }
8685 }

8687 /*
8688  * These routines manipulate the target's queue of pending requests
8689  */
8690 void
8691 mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8692 {
8693     NDBG7(("mptsas_waitq_add: cmd=0x%p", (void *)cmd));
8694     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8695     cmd->cmd_queued = TRUE;
8696     if (ptgt)
8697         ptgt->m_t_nwait++;
8698     if (cmd->cmd_pkt_flags & FLAG_HEAD) {
8699         if ((cmd->cmd_linkp = mpt->m_waitq) == NULL) {
8700             mpt->m_waitqtail = &cmd->cmd_linkp;
8701         }
8702         mpt->m_waitq = cmd;
8703     } else {
8704         cmd->cmd_linkp = NULL;
8705         *(mpt->m_waitqtail) = cmd;
8706         mpt->m_waitqtail = &cmd->cmd_linkp;
8707     }
8708 }

8710 static mptsas_cmd_t *
8711 mptsas_waitq_rm(mptsas_t *mpt)
8712 {
8713     mptsas_cmd_t *cmd;
8714     mptsas_target_t *ptgt;
8715     NDBG7(("mptsas_waitq_rm"));

8717     MPTSAS_WAITQ_RM(mpt, cmd);

8719     NDBG7(("mptsas_waitq_rm: cmd=0x%p", (void *)cmd));
8720     if (cmd) {
8721         ptgt = cmd->cmd_tgt_addr;
8722         if (ptgt) {
8723             ptgt->m_t_nwait--;
8724             ASSERT(ptgt->m_t_nwait >= 0);
8725         }
8726     }
8727     return (cmd);
8728 }

8730 /*
8731  * remove specified cmd from the middle of the wait queue.
8732  */
8733 static void
8734 mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8735 {
8736     mptsas_cmd_t *prevp = mpt->m_waitq;
8737     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

```

```

8739     NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8740         (void *)mpt, (void *)cmd));
8741     if (ptgt) {
8742         ptgt->m_t_nwait--;
8743         ASSERT(ptgt->m_t_nwait >= 0);
8744     }

8746     if (prevp == cmd) {
8747         if ((mpt->m_waitq = cmd->cmd_linkp) == NULL)
8748             mpt->m_waitqtail = &mpt->m_waitq;

8750         cmd->cmd_linkp = NULL;
8751         cmd->cmd_queued = FALSE;
8752         NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8753             (void *)mpt, (void *)cmd));
8754         return;
8755     }

8757     while (prevp != NULL) {
8758         if (prevp->cmd_linkp == cmd) {
8759             if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8760                 mpt->m_waitqtail = &prevp->cmd_linkp;

8762             cmd->cmd_linkp = NULL;
8763             cmd->cmd_queued = FALSE;
8764             NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8765                 (void *)mpt, (void *)cmd));
8766             return;
8767         }
8768         prevp = prevp->cmd_linkp;
8769     }
8770     cmn_err(CE_PANIC, "mpt: mptsas_waitq_delete: queue botch");
8771 }

8773 static mptsas_cmd_t *
8774 mptsas_tx_waitq_rm(mptsas_t *mpt)
8775 {
8776     mptsas_cmd_t *cmd;
8777     NDBG7(("mptsas_tx_waitq_rm"));

8779     MPTSAS_TX_WAITQ_RM(mpt, cmd);

8781     NDBG7(("mptsas_tx_waitq_rm: cmd=0x%p", (void *)cmd));

8783     return (cmd);
8784 }

8786 /*
8787  * remove specified cmd from the middle of the tx_waitq.
8788  */
8789 static void
8790 mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8791 {
8792     mptsas_cmd_t *prevp = mpt->m_tx_waitq;

8794     NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8795         (void *)mpt, (void *)cmd));

8797     if (prevp == cmd) {
8798         if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL)
8799             mpt->m_tx_waitqtail = &mpt->m_tx_waitq;

8801         cmd->cmd_linkp = NULL;
8802         cmd->cmd_queued = FALSE;
8803         NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8804             (void *)mpt, (void *)cmd));

```

```

8805         return;
8806     }

8808     while (prevp != NULL) {
8809         if (prevp->cmd_linkp == cmd) {
8810             if ((prevp->cmd_linkp == cmd->cmd_linkp) == NULL)
8811                 mpt->m_tx_waitqtail = &prevp->cmd_linkp;

8813                 cmd->cmd_linkp = NULL;
8814                 cmd->cmd_queued = FALSE;
8815                 NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8816                     (void *)mpt, (void *)cmd));
8817                 return;
8818             }
8819             prevp = prevp->cmd_linkp;
8820         }
8821         cmn_err(CE_PANIC, "mpt: mptsas_tx_waitq_delete: queue botch");
8822     }

8824 /*
8825  * device and bus reset handling
8826  */
8827 * Notes:
8828 * - RESET_ALL: reset the controller
8829 * - RESET_TARGET: reset the target specified in scsi_address
8830 */
8831 static int
8832 mptsas_scsi_reset(struct scsi_address *ap, int level)
8833 {
8834     mptsas_t          *mpt = ADDR2MPT(ap);
8835     int               rval;
8836     mptsas_tgt_private_t *tgt_private;
8837     mptsas_target_t   *ptgt = NULL;

8839     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->tran_tgt_private;
8840     ptgt = tgt_private->t_private;
8841     if (ptgt == NULL) {
8842         return (FALSE);
8843     }
8844     NDBG22(("mptsas_scsi_reset: target=%d level=%d", ptgt->m_devhdl,
8845         level));

8847     mutex_enter(&mpt->m_mutex);
8848     /*
8849     * if we are not in panic set up a reset delay for this target
8850     */
8851     if (!ddi_in_panic()) {
8852         mptsas_setup_bus_reset_delay(mpt);
8853     } else {
8854         drv_usecwait(mpt->m_scsi_reset_delay * 1000);
8855     }
8856     rval = mptsas_do_scsi_reset(mpt, ptgt->m_devhdl);
8857     mutex_exit(&mpt->m_mutex);

8859     /*
8860     * The transport layer expect to only see TRUE and
8861     * FALSE. Therefore, we will adjust the return value
8862     * if mptsas_do_scsi_reset returns FAILED.
8863     */
8864     if (rval == FAILED)
8865         rval = FALSE;
8866     return (rval);
8867 }

8869 static int
8870 mptsas_do_scsi_reset(mptsas_t *mpt, uint16_t devhdl)

```

```

8871 {
8872     int             rval = FALSE;
8873     uint8_t         config, disk;

8875     ASSERT(mutex_owned(&mpt->m_mutex));

8877     if (mptsas_debug_resets) {
8878         mptsas_log(mpt, CE_WARN, "mptsas_do_scsi_reset: target=%d",
8879             devhdl);
8880     }

8882     /*
8883     * Issue a Target Reset message to the target specified but not to a
8884     * disk making up a raid volume. Just look through the RAID config
8885     * Phys Disk list of DevHandles. If the target's DevHandle is in this
8886     * list, then don't reset this target.
8887     */
8888     for (config = 0; config < mpt->m_num_raid_configs; config++) {
8889         for (disk = 0; disk < MPTSAS_MAX_DISKS_IN_CONFIG; disk++) {
8890             if (devhdl == mpt->m_raidconfig[config].
8891                 m_physdisk_devhdl[disk]) {
8892                 return (TRUE);
8893             }
8894         }
8895     }

8897     rval = mptsas_ioc_task_management(mpt,
8898         MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET, devhdl, 0, NULL, 0, 0);

8900     mptsas_doneq_empty(mpt);
8901     return (rval);
8902 }

8904 static int
8905 mptsas_scsi_reset_notify(struct scsi_address *ap, int flag,
8906     void (*callback)(caddr_t), caddr_t arg)
8907 {
8908     mptsas_t          *mpt = ADDR2MPT(ap);

8910     NDBG22(("mptsas_scsi_reset_notify: tgt=%d", ap->a_target));

8912     return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
8913         &mpt->m_mutex, &mpt->m_reset_notify_listf));
8914 }

8916 static int
8917 mptsas_get_name(struct scsi_device *sd, char *name, int len)
8918 {
8919     dev_info_t         *lun_dip = NULL;

8921     ASSERT(sd != NULL);
8922     ASSERT(name != NULL);
8923     lun_dip = sd->sd_dev;
8924     ASSERT(lun_dip != NULL);

8926     if (mptsas_name_child(lun_dip, name, len) == DDI_SUCCESS) {
8927         return (1);
8928     } else {
8929         return (0);
8930     }
8931 }

8933 static int
8934 mptsas_get_bus_addr(struct scsi_device *sd, char *name, int len)
8935 {
8936     return (mptsas_get_name(sd, name, len));

```

```

8937 }
8939 void
8940 mptsas_set_throttle(mptsas_t *mpt, mptsas_target_t *ptgt, int what)
8941 {
8943     NDBG25(("mptsas_set_throttle: throttle=%x", what));
8945     /*
8946      * if the bus is draining/quiesced, no changes to the throttles
8947      * are allowed. Not allowing change of throttles during draining
8948      * limits error recovery but will reduce draining time
8949      *
8950      * all throttles should have been set to HOLD_THROTTLE
8951      */
8952     if (mpt->m_softstate & (MPTSAS_SS_QUIESCED | MPTSAS_SS_DRAINING)) {
8953         return;
8954     }
8956     if (what == HOLD_THROTTLE) {
8957         ptgt->m_t_throttle = HOLD_THROTTLE;
8958     } else if (ptgt->m_reset_delay == 0) {
8959         ptgt->m_t_throttle = what;
8960     }
8961 }
8963 /*
8964  * Clean up from a device reset.
8965  * For the case of target reset, this function clears the waitq of all
8966  * commands for a particular target. For the case of abort task set, this
8967  * function clears the waitq of all commands for a particular target/lun.
8968  */
8969 static void
8970 mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
8971 {
8972     mptsas_slots_t *slots = mpt->m_active;
8973     mptsas_cmd_t *cmd, *next_cmd;
8974     int slot;
8975     uchar_t reason;
8976     uint_t stat;
8977     hrtime_t timestamp;
8979     NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));
8981     timestamp = gethrtime();
8983     /*
8984      * Make sure the I/O Controller has flushed all cmds
8985      * that are associated with this target for a target reset
8986      * and target/lun for abort task set.
8987      * Account for TM requests, which use the last SMID.
8988      */
8989     for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
8990         if ((cmd = slots->m_slot[slot]) == NULL)
8991             continue;
8992         reason = CMD_RESET;
8993         stat = STAT_DEV_RESET;
8994         switch (tasktype) {
8995             case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8996                 if (Tgt(cmd) == target) {
8997                     if (cmd->cmd_active_expiration <= timestamp) {
8998                         /*
8999                          * When timeout requested, propagate
9000                          * proper reason and statistics to
9001                          * target drivers.
9002                          */

```

```

9003         reason = CMD_TIMEOUT;
9004         stat |= STAT_TIMEOUT;
9005     }
9006     NDBG25(("mptsas_flush_target discovered non-
9007     \"NULL cmd in slot %d, tasktype 0x%x\", slot,
9008     tasktype));
9009     mptsas_dump_cmd(mpt, cmd);
9010     mptsas_remove_cmd(mpt, cmd);
9011     mptsas_set_pkt_reason(mpt, cmd, reason, stat);
9012     mptsas_doneq_add(mpt, cmd);
9013 }
9014 break;
9015 case MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK_SET:
9016     reason = CMD_ABORTED;
9017     stat = STAT_ABORTED;
9018     /*FALLTHROUGH*/
9019 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
9020     if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9022         NDBG25(("mptsas_flush_target discovered non-
9023         \"NULL cmd in slot %d, tasktype 0x%x\", slot,
9024         tasktype));
9025         mptsas_dump_cmd(mpt, cmd);
9026         mptsas_remove_cmd(mpt, cmd);
9027         mptsas_set_pkt_reason(mpt, cmd, reason,
9028         stat);
9029         mptsas_doneq_add(mpt, cmd);
9030     }
9031     break;
9032 default:
9033     break;
9034 }
9035 }
9037 /*
9038  * Flush the waitq and tx_waitq of this target's cmds
9039  */
9040 cmd = mpt->m_waitq;
9042 reason = CMD_RESET;
9043 stat = STAT_DEV_RESET;
9045 switch (tasktype) {
9046 case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
9047     while (cmd != NULL) {
9048         next_cmd = cmd->cmd_linkp;
9049         if (Tgt(cmd) == target) {
9050             mptsas_waitq_delete(mpt, cmd);
9051             mptsas_set_pkt_reason(mpt, cmd,
9052             reason, stat);
9053             mptsas_doneq_add(mpt, cmd);
9054         }
9055         cmd = next_cmd;
9056     }
9057     mutex_enter(&mpt->m_tx_waitq_mutex);
9058     cmd = mpt->m_tx_waitq;
9059     while (cmd != NULL) {
9060         next_cmd = cmd->cmd_linkp;
9061         if (Tgt(cmd) == target) {
9062             mptsas_tx_waitq_delete(mpt, cmd);
9063             mutex_exit(&mpt->m_tx_waitq_mutex);
9064             mptsas_set_pkt_reason(mpt, cmd,
9065             reason, stat);
9066             mptsas_doneq_add(mpt, cmd);
9067             mutex_enter(&mpt->m_tx_waitq_mutex);
9068         }

```

```

9069         cmd = next_cmd;
9070     }
9071     mutex_exit(&mpt->m_tx_waitq_mutex);
9072     break;
9073     case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
9074         reason = CMD_ABORTED;
9075         stat = STAT_ABORTED;
9076         /*FALLTHROUGH*/
9077     case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
9078         while (cmd != NULL) {
9079             next_cmd = cmd->cmd_linkp;
9080             if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9081                 mptsas_waitq_delete(mpt, cmd);
9082                 mptsas_set_pkt_reason(mpt, cmd,
9083                     reason, stat);
9084                 mptsas_doneq_add(mpt, cmd);
9085             }
9086             cmd = next_cmd;
9087         }
9088         mutex_enter(&mpt->m_tx_waitq_mutex);
9089         cmd = mpt->m_tx_waitq;
9090         while (cmd != NULL) {
9091             next_cmd = cmd->cmd_linkp;
9092             if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9093                 mptsas_tx_waitq_delete(mpt, cmd);
9094                 mutex_exit(&mpt->m_tx_waitq_mutex);
9095                 mptsas_set_pkt_reason(mpt, cmd,
9096                     reason, stat);
9097                 mptsas_doneq_add(mpt, cmd);
9098                 mutex_enter(&mpt->m_tx_waitq_mutex);
9099             }
9100             cmd = next_cmd;
9101         }
9102         mutex_exit(&mpt->m_tx_waitq_mutex);
9103         break;
9104     default:
9105         mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
9106             tasktype);
9107         break;
9108     }
9109 }

9111 /*
9112  * Clean up hba state, abort all outstanding command and commands in waitq
9113  * reset timeout of all targets.
9114  */
9115 static void
9116 mptsas_flush_hba(mptsas_t *mpt)
9117 {
9118     mptsas_slots_t *slots = mpt->m_active;
9119     mptsas_cmd_t *cmd;
9120     int slot;

9122     NDBG25(("mptsas_flush_hba"));

9124     /*
9125      * The I/O Controller should have already sent back
9126      * all commands via the scsi I/O reply frame. Make
9127      * sure all commands have been flushed.
9128      * Account for TM request, which use the last SMID.
9129      */
9130     for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
9131         if ((cmd = slots->m_slot[slot]) == NULL)
9132             continue;

9134         if (cmd->cmd_flags & CFLAG_CMDIOC) {

```

```

9135         /*
9136          * Need to make sure to tell everyone that might be
9137          * waiting on this command that it's going to fail. If
9138          * we get here, this command will never timeout because
9139          * the active command table is going to be re-allocated,
9140          * so there will be nothing to check against a time out.
9141          * Instead, mark the command as failed due to reset.
9142          */
9143         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
9144             STAT_BUS_RESET);
9145         if ((cmd->cmd_flags &
9146             (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG))) {
9147             cmd->cmd_flags |= CFLAG_FINISHED;
9148             cv_broadcast(&mpt->m_passthru_cv);
9149             cv_broadcast(&mpt->m_config_cv);
9150             cv_broadcast(&mpt->m_fw_diag_cv);
9151         }
9152         continue;
9153     }

9155     NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
9156         slot));
9157     mptsas_dump_cmd(mpt, cmd);

9159     mptsas_remove_cmd(mpt, cmd);
9160     mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9161     mptsas_doneq_add(mpt, cmd);
9162 }

9164 /*
9165  * Flush the waitq.
9166  */
9167 while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
9168     mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9169     if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
9170         (cmd->cmd_flags & CFLAG_CONFIG) ||
9171         (cmd->cmd_flags & CFLAG_FW_DIAG)) {
9172         cmd->cmd_flags |= CFLAG_FINISHED;
9173         cv_broadcast(&mpt->m_passthru_cv);
9174         cv_broadcast(&mpt->m_config_cv);
9175         cv_broadcast(&mpt->m_fw_diag_cv);
9176     } else {
9177         mptsas_doneq_add(mpt, cmd);
9178     }
9179 }

9181 /*
9182  * Flush the tx_waitq
9183  */
9184 mutex_enter(&mpt->m_tx_waitq_mutex);
9185 while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
9186     mutex_exit(&mpt->m_tx_waitq_mutex);
9187     mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9188     mptsas_doneq_add(mpt, cmd);
9189     mutex_enter(&mpt->m_tx_waitq_mutex);
9190 }
9191 mutex_exit(&mpt->m_tx_waitq_mutex);

9193 /*
9194  * Drain the taskqs prior to reallocating resources.
9195  */
9196 mutex_exit(&mpt->m_mutex);
9197 ddi_taskq_wait(mpt->m_event_taskq);
9198 ddi_taskq_wait(mpt->m_dr_taskq);
9199 mutex_enter(&mpt->m_mutex);
9200 }

```

```

9202 /*
9203  * set pkt_reason and OR in pkt_statistics flag
9204  */
9205 static void
9206 mptsas_set_pkt_reason(mptsas_t *mpt, mptsas_cmd_t *cmd, uchar_t reason,
9207     uint_t stat)
9208 {
9209     #ifndef __lock_lint
9210     _NOTE(ARGUNUSED(mpt))
9211     #endif
9212
9213     NDBG25(("mptsas_set_pkt_reason: cmd=0x%p reason=%x stat=%x",
9214         (void *)cmd, reason, stat));
9215
9216     if (cmd) {
9217         if (cmd->cmd_pkt->pkt_reason == CMD_CMPLT) {
9218             cmd->cmd_pkt->pkt_reason = reason;
9219         }
9220         cmd->cmd_pkt->pkt_statistics |= stat;
9221     }
9222 }
9223
9224 static void
9225 mptsas_start_watch_reset_delay()
9226 {
9227     NDBG22(("mptsas_start_watch_reset_delay"));
9228
9229     mutex_enter(&mptsas_global_mutex);
9230     if (mptsas_reset_watch == NULL && mptsas_timeouts_enabled) {
9231         mptsas_reset_watch = timeout(mptsas_watch_reset_delay, NULL,
9232             drv_usec2hz((clock_t)
9233                 MPTSAS_WATCH_RESET_DELAY_TICK * 1000));
9234         ASSERT(mptsas_reset_watch != NULL);
9235     }
9236     mutex_exit(&mptsas_global_mutex);
9237 }
9238
9239 static void
9240 mptsas_setup_bus_reset_delay(mptsas_t *mpt)
9241 {
9242     mptsas_target_t *ptgt = NULL;
9243
9244     ASSERT(MUTEX_HELD(&mpt->m_mutex));
9245
9246     NDBG22(("mptsas_setup_bus_reset_delay"));
9247     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9248         ptgt = rehash_next(mpt->m_targets, ptgt)) {
9249         mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
9250         ptgt->m_reset_delay = mpt->m_scsi_reset_delay;
9251     }
9252
9253     mptsas_start_watch_reset_delay();
9254 }
9255
9256 /*
9257  * mptsas_watch_reset_delay(_subr) is invoked by timeout() and checks every
9258  * mpt instance for active reset delays
9259  */
9260 static void
9261 mptsas_watch_reset_delay(void *arg)
9262 {
9263     #ifndef __lock_lint
9264     _NOTE(ARGUNUSED(arg))
9265     #endif

```

```

9267     mptsas_t      *mpt;
9268     int           not_done = 0;
9269
9270     NDBG22(("mptsas_watch_reset_delay"));
9271
9272     mutex_enter(&mptsas_global_mutex);
9273     mptsas_reset_watch = 0;
9274     mutex_exit(&mptsas_global_mutex);
9275     rw_enter(&mptsas_global_rwlock, RW_READER);
9276     for (mpt = mptsas_head; mpt != NULL; mpt = mpt->m_next) {
9277         if (mpt->m_tran == 0) {
9278             continue;
9279         }
9280         mutex_enter(&mpt->m_mutex);
9281         not_done += mptsas_watch_reset_delay_subr(mpt);
9282         mutex_exit(&mpt->m_mutex);
9283     }
9284     rw_exit(&mptsas_global_rwlock);
9285
9286     if (not_done) {
9287         mptsas_start_watch_reset_delay();
9288     }
9289 }
9290
9291 static int
9292 mptsas_watch_reset_delay_subr(mptsas_t *mpt)
9293 {
9294     int           done = 0;
9295     int           restart = 0;
9296     mptsas_target_t *ptgt = NULL;
9297
9298     NDBG22(("mptsas_watch_reset_delay_subr: mpt=0x%p", (void *)mpt));
9299
9300     ASSERT(mutex_owned(&mpt->m_mutex));
9301
9302     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9303         ptgt = rehash_next(mpt->m_targets, ptgt)) {
9304         if (ptgt->m_reset_delay != 0) {
9305             ptgt->m_reset_delay -=
9306                 MPTSAS_WATCH_RESET_DELAY_TICK;
9307             if (ptgt->m_reset_delay <= 0) {
9308                 ptgt->m_reset_delay = 0;
9309                 mptsas_set_throttle(mpt, ptgt,
9310                     MAX_THROTTLE);
9311                 restart++;
9312             } else {
9313                 done = -1;
9314             }
9315         }
9316     }
9317
9318     if (restart > 0) {
9319         mptsas_restart_hba(mpt);
9320     }
9321     return (done);
9322 }
9323
9324 #ifdef MPTSAS_TEST
9325 static void
9326 mptsas_test_reset(mptsas_t *mpt, int target)
9327 {
9328     mptsas_target_t *ptgt = NULL;
9329
9330     if (mptsas_rtest == target) {
9331         if (mptsas_do_scsi_reset(mpt, target) == TRUE) {
9332             mptsas_rtest = -1;

```

```

9333     }
9334     if (mptsas_rtest == -1) {
9335         NDBG22(("mptsas_test_reset success"));
9336     }
9337 }
9338 }
9339 #endif

9341 /*
9342  * abort handling:
9343  *
9344  * Notes:
9345  *   - if pkt is not NULL, abort just that command
9346  *   - if pkt is NULL, abort all outstanding commands for target
9347  */
9348 static int
9349 mptsas_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
9350 {
9351     mptsas_t          *mpt = ADDR2MPT(ap);
9352     int                rval;
9353     mptsas_tgt_private_t *tgt_private;
9354     int                target, lun;

9356     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
9357     tran_tgt_private;
9358     ASSERT(tgt_private != NULL);
9359     target = tgt_private->t_private->m_devhdl;
9360     lun = tgt_private->t_lun;

9362     NDBG23(("mptsas_scsi_abort: target=%d.%d", target, lun));

9364     mutex_enter(&mpt->m_mutex);
9365     rval = mptsas_do_scsi_abort(mpt, target, lun, pkt);
9366     mutex_exit(&mpt->m_mutex);
9367     return (rval);
9368 }

9370 static int
9371 mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun, struct scsi_pkt *pkt)
9372 {
9373     mptsas_cmd_t      *sp = NULL;
9374     mptsas_slots_t    *slots = mpt->m_active;
9375     int                rval = FALSE;

9377     ASSERT(mutex_owned(&mpt->m_mutex));

9379     /*
9380     * Abort the command pkt on the target/lun in ap.  If pkt is
9381     * NULL, abort all outstanding commands on that target/lun.
9382     * If you can abort them, return 1, else return 0.
9383     * Each packet that's aborted should be sent back to the target
9384     * driver through the callback routine, with pkt_reason set to
9385     * CMD_ABORTED.
9386     *
9387     * abort cmd pkt on HBA hardware; clean out of outstanding
9388     * command lists, etc.
9389     */
9390     if (pkt != NULL) {
9391         /* abort the specified packet */
9392         sp = PKT2CMD(pkt);

9394         if (sp->cmd_queued) {
9395             NDBG23(("mptsas_do_scsi_abort: queued sp=0x%p aborted",
9396             (void *)sp));
9397             mptsas_waitq_delete(mpt, sp);
9398             mptsas_set_pkt_reason(mpt, sp, CMD_ABORTED,

```

```

9399         STAT_ABORTED);
9400         mptsas_doneq_add(mpt, sp);
9401         rval = TRUE;
9402         goto done;
9403     }

9405     /*
9406     * Have mpt firmware abort this command
9407     */

9409     if (slots->m_slot[sp->cmd_slot] != NULL) {
9410         rval = mptsas_ioc_task_management(mpt,
9411         MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK, target,
9412         lun, NULL, 0, 0);

9414         /*
9415         * The transport layer expects only TRUE and FALSE.
9416         * Therefore, if mptsas_ioc_task_management returns
9417         * FAILED we will return FALSE.
9418         */
9419         if (rval == FAILED)
9420             rval = FALSE;
9421         goto done;
9422     }
9423 }

9425     /*
9426     * If pkt is NULL then abort task set
9427     */
9428     rval = mptsas_ioc_task_management(mpt,
9429     MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK_SET, target, lun, NULL, 0, 0);

9431     /*
9432     * The transport layer expects only TRUE and FALSE.
9433     * Therefore, if mptsas_ioc_task_management returns
9434     * FAILED we will return FALSE.
9435     */
9436     if (rval == FAILED)
9437         rval = FALSE;

9439 #ifdef MPTSAS_TEST
9440     if (rval && mptsas_test_stop) {
9441         debug_enter("mptsas_do_scsi_abort");
9442     }
9443 #endif

9445 done:
9446     mptsas_doneq_empty(mpt);
9447     return (rval);
9448 }

9450 /*
9451  * capability handling:
9452  * (*tran_getcap).  Get the capability named, and return its value.
9453  */
9454 static int
9455 mptsas_scsi_getcap(struct scsi_address *ap, char *cap, int tgtonly)
9456 {
9457     mptsas_t          *mpt = ADDR2MPT(ap);
9458     int                ckey;
9459     int                rval = FALSE;

9461     NDBG24(("mptsas_scsi_getcap: target=%d, cap=%s tgtonly=%x",
9462     ap->a_target, cap, tgtonly));

9464     mutex_enter(&mpt->m_mutex);

```



```

9466     if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9467         mutex_exit(&mpt->m_mutex);
9468         return (UNDEFINED);
9469     }

9471     switch (ckey) {
9472     case SCSI_CAP_DMA_MAX:
9473         rval = (int)mpt->m_msg_dma_attr.dma_attr_maxxfer;
9474         break;
9475     case SCSI_CAP_ARQ:
9476         rval = TRUE;
9477         break;
9478     case SCSI_CAP_MSG_OUT:
9479     case SCSI_CAP_PARITY:
9480     case SCSI_CAP_UNTAGGED_QING:
9481         rval = TRUE;
9482         break;
9483     case SCSI_CAP_TAGGED_QING:
9484         rval = TRUE;
9485         break;
9486     case SCSI_CAP_RESET_NOTIFICATION:
9487         rval = TRUE;
9488         break;
9489     case SCSI_CAP_LINKED_CMDS:
9490         rval = FALSE;
9491         break;
9492     case SCSI_CAP_QFULL_RETRIES:
9493         rval = ((mptsas_tgt_private_t *) (ap->a_hba_tran->
9494             tran_tgt_private))->t_private->m_qfull_retries;
9495         break;
9496     case SCSI_CAP_QFULL_RETRY_INTERVAL:
9497         rval = drv_hztousec(((mptsas_tgt_private_t *)
9498             (ap->a_hba_tran->tran_tgt_private))->
9499             t_private->m_qfull_retry_interval) / 1000;
9500         break;
9501     case SCSI_CAP_CDB_LEN:
9502         rval = CDB_GROUP4;
9503         break;
9504     case SCSI_CAP_INTERCONNECT_TYPE:
9505         rval = INTERCONNECT_SAS;
9506         break;
9507     case SCSI_CAP_TRAN_LAYER_RETRIES:
9508         if (mpt->m_ioc_capabilities &
9509             MPI2_IOCFACETS_CAPABILITY_TLR)
9510             rval = TRUE;
9511         else
9512             rval = FALSE;
9513         break;
9514     default:
9515         rval = UNDEFINED;
9516         break;
9517     }

9519     NDBG24(("mptsas_scsi_getcap: %s, rval=%x", cap, rval));

9521     mutex_exit(&mpt->m_mutex);
9522     return (rval);
9523 }

9525 /*
9526  * (*tran_setcap). Set the capability named to the value given.
9527  */
9528 static int
9529 mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value, int tgtonly)
9530 {

```

```

9531     mptsas_t      *mpt = ADDR2MPT(ap);
9532     int           ckey;
9533     int           rval = FALSE;

9535     NDBG24(("mptsas_scsi_setcap: target=%d, cap=%s value=%x tgtonly=%x",
9536         ap->a_target, cap, value, tgtonly));

9538     if (!tgtonly) {
9539         return (rval);
9540     }

9542     mutex_enter(&mpt->m_mutex);

9544     if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9545         mutex_exit(&mpt->m_mutex);
9546         return (UNDEFINED);
9547     }

9549     switch (ckey) {
9550     case SCSI_CAP_DMA_MAX:
9551     case SCSI_CAP_MSG_OUT:
9552     case SCSI_CAP_PARITY:
9553     case SCSI_CAP_INITIATOR_ID:
9554     case SCSI_CAP_LINKED_CMDS:
9555     case SCSI_CAP_UNTAGGED_QING:
9556     case SCSI_CAP_RESET_NOTIFICATION:
9557         /*
9558          * None of these are settable via
9559          * the capability interface.
9560          */
9561         break;
9562     case SCSI_CAP_ARQ:
9563         /*
9564          * We cannot turn off arq so return false if asked to
9565          */
9566         if (value) {
9567             rval = TRUE;
9568         } else {
9569             rval = FALSE;
9570         }
9571         break;
9572     case SCSI_CAP_TAGGED_QING:
9573         mptsas_set_throttle(mpt, ((mptsas_tgt_private_t *)
9574             (ap->a_hba_tran->tran_tgt_private))->t_private,
9575             MAX_THROTTLE);
9576         rval = TRUE;
9577         break;
9578     case SCSI_CAP_QFULL_RETRIES:
9579         ((mptsas_tgt_private_t *) (ap->a_hba_tran->tran_tgt_private))->
9580             t_private->m_qfull_retries = (uchar_t)value;
9581         rval = TRUE;
9582         break;
9583     case SCSI_CAP_QFULL_RETRY_INTERVAL:
9584         ((mptsas_tgt_private_t *) (ap->a_hba_tran->tran_tgt_private))->
9585             t_private->m_qfull_retry_interval =
9586             drv_usectohz(value * 1000);
9587         rval = TRUE;
9588         break;
9589     default:
9590         rval = UNDEFINED;
9591         break;
9592     }
9593     mutex_exit(&mpt->m_mutex);
9594     return (rval);
9595 }

```

```

9597 /*
9598  * Utility routine for mptsas_ifsetcap/ifgetcap
9599  */
9600 /*ARGSUSED*/
9601 static int
9602 mptsas_scsi_capchk(char *cap, int tgtonly, int *cidxp)
9603 {
9604     NDBG24(("mptsas_scsi_capchk: cap=%s", cap));

9606     if (!cap)
9607         return (FALSE);

9609     *cidxp = scsi_hba_lookup_capstr(cap);
9610     return (TRUE);
9611 }

9613 static int
9614 mptsas_alloc_active_slots(mptsas_t *mpt, int flag)
9615 {
9616     mptsas_slots_t *old_active = mpt->m_active;
9617     mptsas_slots_t *new_active;
9618     size_t size;

9620     /*
9621      * if there are active commands, then we cannot
9622      * change size of active slots array.
9623      */
9624     ASSERT(mpt->m_ncmds == 0);

9626     size = MPTSAS_SLOTS_SIZE(mpt);
9627     new_active = kmem_zalloc(size, flag);
9628     if (new_active == NULL) {
9629         NDBG1(("new active alloc failed"));
9630         return (-1);
9631     }
9632     /*
9633      * Since SMID 0 is reserved and the TM slot is reserved, the
9634      * number of slots that can be used at any one time is
9635      * m_max_requests - 2.
9636      */
9637     new_active->m_n_normal = (mpt->m_max_requests - 2);
9638     new_active->m_size = size;
9639     new_active->m_rotor = 1;
9640     if (old_active)
9641         mptsas_free_active_slots(mpt);
9642     mpt->m_active = new_active;

9644     return (0);
9645 }

9647 static void
9648 mptsas_free_active_slots(mptsas_t *mpt)
9649 {
9650     mptsas_slots_t *active = mpt->m_active;
9651     size_t size;

9653     if (active == NULL)
9654         return;
9655     size = active->m_size;
9656     kmem_free(active, size);
9657     mpt->m_active = NULL;
9658 }

9660 /*
9661  * Error logging, printing, and debug print routines.
9662  */

```

```

9663 static char *mptsas_label = "mpt_sas";

9665 /*PRINTFLIKE3*/
9666 void
9667 mptsas_log(mptsas_t *mpt, int level, char *fmt, ...)
9668 {
9669     dev_info_t *dev;
9670     va_list ap;

9672     if (mpt) {
9673         dev = mpt->m_dip;
9674     } else {
9675         dev = 0;
9676     }

9678     mutex_enter(&mptsas_log_mutex);

9680     va_start(ap, fmt);
9681     (void) vsprintf(mptsas_log_buf, fmt, ap);
9682     va_end(ap);

9684     if (level == CE_CONT) {
9685         scsi_log(dev, mptsas_label, level, "%s\n", mptsas_log_buf);
9686     } else {
9687         scsi_log(dev, mptsas_label, level, "%s", mptsas_log_buf);
9688     }

9690     mutex_exit(&mptsas_log_mutex);
9691 }

9693 #ifdef MPTSAS_DEBUG
9694 /*
9695  * Use a circular buffer to log messages to private memory.
9696  * Increment idx atomically to minimize risk to miss lines.
9697  * It's fast and does not hold up the proceedings too much.
9698  */
9699 static const size_t mptsas_dbglog_linecnt = MPTSAS_DBGLOG_LINECNT;
9700 static const size_t mptsas_dbglog_linelen = MPTSAS_DBGLOG_LINELEN;
9701 static char mptsas_dbglog_bufs[MPTSAS_DBGLOG_LINECNT][MPTSAS_DBGLOG_LINELEN];
9702 static uint32_t mptsas_dbglog_idx = 0;

9704 /*PRINTFLIKE1*/
9705 void
9706 mptsas_debug_log(char *fmt, ...)
9707 {
9708     va_list ap;
9709     uint32_t idx;

9711     idx = atomic_inc_32_nv(&mptsas_dbglog_idx) &
9712         (mptsas_dbglog_linecnt - 1);

9714     va_start(ap, fmt);
9715     (void) vsnprintf(mptsas_dbglog_bufs[idx],
9716         mptsas_dbglog_linelen, fmt, ap);
9717     va_end(ap);
9718 }

9720 /*PRINTFLIKE1*/
9721 void
9722 mptsas_printf(char *fmt, ...)
9723 {
9724     dev_info_t *dev = 0;
9725     va_list ap;

9727     mutex_enter(&mptsas_log_mutex);

```

```

9729     va_start(ap, fmt);
9730     (void) vsprintf(mptsas_log_buf, fmt, ap);
9731     va_end(ap);

9733 #ifndef PROM_PRINTF
9734     prom_printf("%s:\t%s\n", mptsas_label, mptsas_log_buf);
9735 #else
9736     scsi_log(dev, mptsas_label, CE_CONT, "!%s\n", mptsas_log_buf);
9737 #endif
9738     mutex_exit(&mptsas_log_mutex);
9739 }
9740 #endif

9742 /*
9743  * timeout handling
9744  */
9745 static void
9746 mptsas_watch(void *arg)
9747 {
9748 #ifndef __lock_lint
9749     _NOTE(ARGUNUSED(arg))
9750 #endif

9752     mptsas_t      *mpt;
9753     uint32_t      doorbell;

9755     NDBG30(("mptsas_watch"));

9757     rw_enter(&mptsas_global_rwlock, RW_READER);
9758     for (mpt = mptsas_head; mpt != (mptsas_t *)NULL; mpt = mpt->m_next) {

9760         mutex_enter(&mpt->m_mutex);

9762         /* Skip device if not powered on */
9763         if (mpt->m_options & MPTSAS_OPT_PM) {
9764             if (mpt->m_power_level == PM_LEVEL_D0) {
9765                 (void) pm_busy_component(mpt->m_dip, 0);
9766                 mpt->m_busy = 1;
9767             } else {
9768                 mutex_exit(&mpt->m_mutex);
9769                 continue;
9770             }
9771         }

9773         /*
9774          * Check if controller is in a FAULT state. If so, reset it.
9775          */
9776         doorbell = ddi_get32(mpt->m_datap, &mpt->m_reg->Doorbell);
9777         if ((doorbell & MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_FAULT) {
9778             doorbell &= MPI2_DOORBELL_DATA_MASK;
9779             mptsas_log(mpt, CE_WARN, "MPT Firmware Fault, "
9780                 "code: %04x", doorbell);
9781             mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
9782             if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
9783                 mptsas_log(mpt, CE_WARN, "Reset failed"
9784                     "after fault was detected");
9785             }
9786         }

9788         /*
9789          * For now, always call mptsas_watchsubr.
9790          */
9791         mptsas_watchsubr(mpt);

9793         if (mpt->m_options & MPTSAS_OPT_PM) {
9794             mpt->m_busy = 0;

```

```

9795         (void) pm_idle_component(mpt->m_dip, 0);
9796     }

9798     mutex_exit(&mpt->m_mutex);
9799 }
9800 rw_exit(&mptsas_global_rwlock);

9802     mutex_enter(&mptsas_global_mutex);
9803     if (mptsas_timeouts_enabled)
9804         mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
9805     mutex_exit(&mptsas_global_mutex);
9806 }

9808 static void
9809 mptsas_watchsubr(mptsas_t *mpt)
9810 {
9811     int            i;
9812     mptsas_cmd_t  *cmd;
9813     mptsas_target_t *ptgt = NULL;
9814     hrtime_t      timestamp = gethrtime();

9816     ASSERT(MUTEX_HELD(&mpt->m_mutex));

9818     NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

9820 #ifndef MPTSAS_TEST
9821     if (mptsas_enable_untagged) {
9822         mptsas_test_untagged++;
9823     }
9824 #endif

9826     /*
9827      * Check for commands stuck in active slot
9828      * Account for TM requests, which use the last SMID.
9829      */
9830     for (i = 0; i <= mpt->m_active->m_n_normal; i++) {
9831         if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9832             if (cmd->cmd_active_expiration <= timestamp) {
9833                 if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9834                     /*
9835                      * There seems to be a command stuck
9836                      * in the active slot. Drain throttle.
9837                      */
9838                     mptsas_set_throttle(mpt,
9839                         cmd->cmd_tgt_addr,
9840                         DRAIN_THROTTLE);
9841                 } else if (cmd->cmd_flags &
9842                     (CFLAG_PASSTHRU | CFLAG_CONFIG |
9843                     CFLAG_FW_DIAG)) {
9844                     /*
9845                      * passthrough command timeout
9846                      */
9847                     cmd->cmd_flags |= (CFLAG_FINISHED |
9848                         CFLAG_TIMEOUT);
9849                     cv_broadcast(&mpt->m_passthru_cv);
9850                     cv_broadcast(&mpt->m_config_cv);
9851                     cv_broadcast(&mpt->m_fw_diag_cv);
9852                 }
9853             }
9854         }
9855     }

9857     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9858         ptgt = rehash_next(mpt->m_targets, ptgt)) {
9859         /*
9860          * If we were draining due to a qfull condition,

```

```

9861     * go back to full throttle.
9862     */
9863     if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9864         (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9865         (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9866         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9867         mptsas_restart_hba(mpt);
9868     }

9870     cmd = TAILQ_LAST(&ptgt->m_active_cmdq, mptsas_active_cmdq);
9871     if (cmd == NULL)
9872         continue;

9874     if (cmd->cmd_active_expiration <= timestamp) {
9875         /*
9876          * Earliest command timeout expired. Drain throttle.
9877          */
9878         mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);

9880         /*
9881          * Check for remaining commands.
9882          */
9883         cmd = TAILQ_FIRST(&ptgt->m_active_cmdq);
9884         if (cmd->cmd_active_expiration > timestamp) {
9885             /*
9886              * Wait for remaining commands to complete or
9887              * time out.
9888              */
9889             NDBG23(("command timed out, pending drain"));
9890             continue;
9891         }

9893         /*
9894          * All command timeouts expired.
9895          */
9896         mptsas_log(mpt, CE_NOTE, "Timeout of %d seconds "
9897                  "expired with %d commands on target %d lun %d.",
9898                  cmd->cmd_pkt->pkt_time, ptgt->m_t_ncmds,
9899                  ptgt->m_devhdl, Lun(cmd));

9901         mptsas_cmd_timeout(mpt, ptgt);
9902     } else if (cmd->cmd_active_expiration <=
9903               timestamp + (hrtime_t)mptsas_scsi_watchdog_tick * NANOSEC) {
9904         NDBG23(("pending timeout"));
9905         mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
9906     }
9907 }
9908 }

9910 /*
9911  * timeout recovery
9912  */
9913 static void
9914 mptsas_cmd_timeout(mptsas_t *mpt, mptsas_target_t *ptgt)
9915 {
9916     uint16_t     devhdl;
9917     uint64_t     sas_wwn;
9918     uint8_t      phy;
9919     char         wwn_str[MPTSAS_WWN_STRLEN];

9921     devhdl = ptgt->m_devhdl;
9922     sas_wwn = ptgt->m_addr.mta_wwn;
9923     phy = ptgt->m_phynum;
9924     if (sas_wwn == 0) {
9925         (void) sprintf(wwn_str, "p%x", phy);
9926     } else {

```

```

9927         (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
9928     }

9930     NDBG29(("mptsas_cmd_timeout: target=%d", devhdl));
9931     mptsas_log(mpt, CE_WARN, "Disconnected command timeout for "
9932              "target %d %s, enclosure %u", devhdl, wwn_str,
9933              ptgt->m_enclosure);

9935     /*
9936      * Abort all outstanding commands on the device.
9937      */
9938     NDBG29(("mptsas_cmd_timeout: device reset"));
9939     if (mptsas_do_scsi_reset(mpt, devhdl) != TRUE) {
9940         mptsas_log(mpt, CE_WARN, "Target %d reset for command timeout "
9941                  "recovery failed!", devhdl);
9942     }
9943 }

9945 /*
9946  * Device / Hotplug control
9947  */
9948 static int
9949 mptsas_scsi_quiesce(dev_info_t *dip)
9950 {
9951     mptsas_t     *mpt;
9952     scsi_hba_tran_t *tran;

9954     tran = ddi_get_driver_private(dip);
9955     if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
9956         return (-1);

9958     return (mptsas_quiesce_bus(mpt));
9959 }

9961 static int
9962 mptsas_scsi_unquiesce(dev_info_t *dip)
9963 {
9964     mptsas_t     *mpt;
9965     scsi_hba_tran_t *tran;

9967     tran = ddi_get_driver_private(dip);
9968     if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
9969         return (-1);

9971     return (mptsas_unquiesce_bus(mpt));
9972 }

9974 static int
9975 mptsas_quiesce_bus(mptsas_t *mpt)
9976 {
9977     mptsas_target_t *ptgt = NULL;

9979     NDBG28(("mptsas_quiesce_bus"));
9980     mutex_enter(&mpt->m_mutex);

9982     /* Set all the throttles to zero */
9983     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9984          ptgt = rehash_next(mpt->m_targets, ptgt)) {
9985         mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
9986     }

9988     /* If there are any outstanding commands in the queue */
9989     if (mpt->m_ncmds) {
9990         mpt->m_softstate |= MPTSAS_SS_DRAINING;
9991         mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
9992         mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectoh(1000000)));

```

```

9993     if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
9994         /*
9995          * Quiesce has been interrupted
9996          */
9997         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
9998         for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9999              ptgt = rehash_next(mpt->m_targets, ptgt)) {
10000             mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10001         }
10002         mptsas_restart_hba(mpt);
10003         if (mpt->m_quiesce_timeid != 0) {
10004             timeout_id_t tid = mpt->m_quiesce_timeid;
10005             mpt->m_quiesce_timeid = 0;
10006             mutex_exit(&mpt->m_mutex);
10007             (void) untimedout(tid);
10008             return (-1);
10009         }
10010         mutex_exit(&mpt->m_mutex);
10011         return (-1);
10012     } else {
10013         /* Bus has been quiesced */
10014         ASSERT(mpt->m_quiesce_timeid == 0);
10015         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10016         mpt->m_softstate |= MPTSAS_SS_QUIESCED;
10017         mutex_exit(&mpt->m_mutex);
10018         return (0);
10019     }
10020 }
10021 /* Bus was not busy - QUIESCED */
10022 mutex_exit(&mpt->m_mutex);
10023
10024 return (0);
10025 }
10026
10027 static int
10028 mptsas_unquiesce_bus(mptsas_t *mpt)
10029 {
10030     mptsas_target_t *ptgt = NULL;
10031
10032     NDBG28(("mptsas_unquiesce_bus"));
10033     mutex_enter(&mpt->m_mutex);
10034     mpt->m_softstate &= ~MPTSAS_SS_QUIESCED;
10035     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
10036         ptgt = rehash_next(mpt->m_targets, ptgt)) {
10037         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10038     }
10039     mptsas_restart_hba(mpt);
10040     mutex_exit(&mpt->m_mutex);
10041     return (0);
10042 }
10043
10044 static void
10045 mptsas_ncmds_checkdrain(void *arg)
10046 {
10047     mptsas_t *mpt = arg;
10048     mptsas_target_t *ptgt = NULL;
10049
10050     mutex_enter(&mpt->m_mutex);
10051     if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
10052         mpt->m_quiesce_timeid = 0;
10053         if (mpt->m_ncmds == 0) {
10054             /* Command queue has been drained */
10055             cv_signal(&mpt->m_cv);
10056         } else {
10057             /*
10058              * The throttle may have been reset because

```

```

10059         * of a SCSI bus reset
10060         */
10061         for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
10062             ptgt = rehash_next(mpt->m_targets, ptgt)) {
10063             mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10064         }
10065
10066         mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10067             mpt, (MPTSAS_QUIESCE_TIMEOUT *
10068                 drv_usec2hz(1000000)));
10069     }
10070 }
10071 mutex_exit(&mpt->m_mutex);
10072 }
10073
10074 /*ARGSUSED*/
10075 static void
10076 mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
10077 {
10078     int i;
10079     uint8_t *cp = (uchar_t *)cmd->cmd_pkt->pkt_cdbp;
10080     char buf[128];
10081
10082     buf[0] = '\0';
10083     NDBG25(("?Cmd (0x%p) dump for Target %d Lun %d:\n", (void *)cmd,
10084         Tgt(cmd), Lun(cmd)));
10085     (void) sprintf(&buf[0], "\tcdb=[");
10086     for (i = 0; i < (int)cmd->cmd_cdblen; i++) {
10087         (void) sprintf(&buf[strlen(buf)], " 0x%x", *cp++);
10088     }
10089     (void) sprintf(&buf[strlen(buf)], " ]");
10090     NDBG25(("?%s\n", buf));
10091     NDBG25(("?pkt flags=0x%x pkt_statistics=0x%x pkt_state=0x%x\n",
10092         cmd->cmd_pkt->pkt_flags, cmd->cmd_pkt->pkt_statistics,
10093         cmd->cmd_pkt->pkt_state));
10094     NDBG25(("?pkt_scbp=0x%x cmd_flags=0x%x\n", cmd->cmd_pkt->pkt_scbp ?
10095         *(cmd->cmd_pkt->pkt_scbp) : 0, cmd->cmd_flags));
10096 }
10097
10098 static void
10099 mptsas_passthru_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10100     pMpi2SGESimple64_t sgep)
10101 {
10102     uint32_t sge_flags;
10103     uint32_t data_size, dataout_size;
10104     ddi_dma_cookie_t data_cookie;
10105     ddi_dma_cookie_t dataout_cookie;
10106
10107     data_size = pt->data_size;
10108     dataout_size = pt->dataout_size;
10109     data_cookie = pt->data_cookie;
10110     dataout_cookie = pt->dataout_cookie;
10111
10112     if (dataout_size) {
10113         sge_flags = dataout_size |
10114             ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10115                 MPI2_SGE_FLAGS_END_OF_BUFFER |
10116                 MPI2_SGE_FLAGS_HOST_TO_IOC |
10117                 MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10118             MPI2_SGE_FLAGS_SHIFT);
10119         ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
10120         ddi_put32(acc_hdl, &sgep->Address.Low,
10121             (uint32_t)(dataout_cookie.dmac_laddress &
10122                 0xffffffff));
10123         ddi_put32(acc_hdl, &sgep->Address.High,
10124             (uint32_t)(dataout_cookie.dmac_laddress

```

```

10125         >> 32));
10126         sgep++;
10127     }
10128     sge_flags = data_size;
10129     sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10130     MPI2_SGE_FLAGS_LAST_ELEMENT |
10131     MPI2_SGE_FLAGS_END_OF_BUFFER |
10132     MPI2_SGE_FLAGS_END_OF_LIST |
10133     MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10134     MPI2_SGE_FLAGS_SHIFT);
10135     if (pt->direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10136         sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
10137         MPI2_SGE_FLAGS_SHIFT);
10138     } else {
10139         sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
10140         MPI2_SGE_FLAGS_SHIFT);
10141     }
10142     ddi_put32(acc_hdl, &sgep->FlagsLength,
10143     sge_flags);
10144     ddi_put32(acc_hdl, &sgep->Address.Low,
10145     (uint32_t)(data_cookie.dmac_laddress &
10146     0xffffffffull));
10147     ddi_put32(acc_hdl, &sgep->Address.High,
10148     (uint32_t)(data_cookie.dmac_laddress >> 32));
10149 }

10151 static void
10152 mptsas_passthru_ieee_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10153     pMpi2IeeeSgeSimple64_t ieeeesgep)
10154 {
10155     uint8_t         sge_flags;
10156     uint32_t        data_size, dataout_size;
10157     ddi_dma_cookie_t data_cookie;
10158     ddi_dma_cookie_t dataout_cookie;

10160     data_size = pt->data_size;
10161     dataout_size = pt->dataout_size;
10162     data_cookie = pt->data_cookie;
10163     dataout_cookie = pt->dataout_cookie;

10165     sge_flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
10166     MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
10167     if (dataout_size) {
10168         ddi_put32(acc_hdl, &ieeeesgep->Length, dataout_size);
10169         ddi_put32(acc_hdl, &ieeeesgep->Address.Low,
10170         (uint32_t)(dataout_cookie.dmac_laddress &
10171         0xffffffffull));
10172         ddi_put32(acc_hdl, &ieeeesgep->Address.High,
10173         (uint32_t)(dataout_cookie.dmac_laddress >> 32));
10174         ddi_put8(acc_hdl, &ieeeesgep->Flags, sge_flags);
10175         ieeeesgep++;
10176     }
10177     sge_flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
10178     ddi_put32(acc_hdl, &ieeeesgep->Length, data_size);
10179     ddi_put32(acc_hdl, &ieeeesgep->Address.Low,
10180     (uint32_t)(data_cookie.dmac_laddress & 0xffffffffull));
10181     ddi_put32(acc_hdl, &ieeeesgep->Address.High,
10182     (uint32_t)(data_cookie.dmac_laddress >> 32));
10183     ddi_put8(acc_hdl, &ieeeesgep->Flags, sge_flags);
10184 }

10186 static void
10187 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
10188 {
10189     caddr_t         memp;
10190     pMPI2RequestHeader_t request_hdrp;

```

```

10191     struct scsi_pkt      *pkt = cmd->cmd_pkt;
10192     mptsas_pt_request_t  *pt = pkt->pkt_ha_private;
10193     uint32_t             request_size;
10194     uint32_t             request_desc_low, request_desc_high = 0;
10195     uint32_t             i, sense_bufp;
10196     uint8_t             desc_type;
10197     uint8_t             *request, function;
10198     ddi_dma_handle_t     dma_hdl = mpt->m_dma_req_frame_hdl;
10199     ddi_acc_handle_t     acc_hdl = mpt->m_acc_req_frame_hdl;

10201     desc_type = MPI2_REQ_DESCRIPTOR_FLAGS_DEFAULT_TYPE;

10203     request = pt->request;
10204     request_size = pt->request_size;

10206     /*
10207     * Store the passthrough message in memory location
10208     * corresponding to our slot number
10209     */
10210     memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
10211     request_hdrp = (pMPI2RequestHeader_t)memp;
10212     bzero(memp, mpt->m_req_frame_size);

10214     for (i = 0; i < request_size; i++) {
10215         bcopy(request + i, memp + i, 1);
10216     }

10218     NDBG15(("mptsas_start_passthru: Func 0x%x, MsgFlags 0x%x, "
10219     "size=%d, in %d, out %d", request_hdrp->Function,
10220     request_hdrp->MsgFlags, request_size,
10221     pt->data_size, pt->dataout_size));

10223     /*
10224     * Add an SGE, even if the length is zero.
10225     */
10226     if (mpt->m_MPI25 && pt->simple == 0) {
10227         mptsas_passthru_ieee_sge(acc_hdl, pt,
10228         (pMpi2IeeeSgeSimple64_t)
10229         ((uint8_t *)request_hdrp + pt->sgl_offset));
10230     } else {
10231         mptsas_passthru_sge(acc_hdl, pt,
10232         (pMpi2SGESimple64_t)
10233         ((uint8_t *)request_hdrp + pt->sgl_offset));
10234     }

10236     function = request_hdrp->Function;
10237     if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10238     (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10239         pMpi2SCSIIORequest_t scsi_io_req;

10241         NDBG15(("mptsas_start_passthru: Is SCSI IO Req"));
10242         scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;
10243         /*
10244         * Put SGE for data and data_out buffer at the end of
10245         * scsi_io_request message header.(64 bytes in total)
10246         * Following above SGEs, the residual space will be
10247         * used by sense data.
10248         */
10249         ddi_put8(acc_hdl,
10250         &scsi_io_req->SenseBufferLength,
10251         (uint8_t)(request_size - 64));

10253         sense_bufp = mpt->m_req_frame_dma_addr +
10254         (mpt->m_req_frame_size * cmd->cmd_slot);
10255         sense_bufp += 64;
10256         ddi_put32(acc_hdl,

```

```

10257     &scsi_io_req->SenseBufferLowAddress, sense_bufp);
10259
10260     /*
10261     * Set SGLOffset0 value
10262     */
10263     ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
10264             offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);
10265
10266     /*
10267     * Setup descriptor info. RAID passthrough must use the
10268     * default request descriptor which is already set, so if this
10269     * is a SCSI IO request, change the descriptor to SCSI IO.
10270     */
10271     if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
10272         desc_type = MPI2_REQ_DESCRIPTOR_FLAGS_SCSI_IO;
10273         request_desc_high = (ddi_get16(acc_hdl,
10274             &scsi_io_req->DevHandle) << 16);
10275     }
10276
10277     /*
10278     * We must wait till the message has been completed before
10279     * beginning the next message so we wait for this one to
10280     * finish.
10281     */
10282     (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
10283     request_desc_low = (cmd->cmd_slot << 16) + desc_type;
10284     cmd->cmd_rfm = NULL;
10285     MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
10286     if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
10287         (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
10288         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10289     }
10290 }
10291
10292 typedef void (mptsas_pre_f)(mptsas_t *, mptsas_pt_request_t *);
10293 static mptsas_pre_f mpi_pre_ioc_facts;
10294 static mptsas_pre_f mpi_pre_port_facts;
10295 static mptsas_pre_f mpi_pre_fw_download;
10296 static mptsas_pre_f mpi_pre_fw_25_download;
10297 static mptsas_pre_f mpi_pre_fw_upload;
10298 static mptsas_pre_f mpi_pre_fw_25_upload;
10299 static mptsas_pre_f mpi_pre_sata_passthrough;
10300 static mptsas_pre_f mpi_pre_smp_passthrough;
10301 static mptsas_pre_f mpi_pre_config;
10302 static mptsas_pre_f mpi_pre_sas_io_unit_control;
10303 static mptsas_pre_f mpi_pre_scsi_io_req;
10304
10305 /*
10306 * Prepare the pt for a SAS2 FW_DOWNLOAD request.
10307 */
10308 static void
10309 mpi_pre_fw_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10310 {
10311     pMpi2FWDownloadTCSGE_t tcsge;
10312     pMpi2FWDownloadRequest req;
10313
10314     /*
10315     * If SAS3, call separate function.
10316     */
10317     if (mpt->m_MPI25) {
10318         mpi_pre_fw_25_download(mpt, pt);
10319     }
10320 }
10321
10322 /*

```

```

10323     * User requests should come in with the Transaction
10324     * context element where the SGL will go. Putting the
10325     * SGL after that seems to work, but don't really know
10326     * why. Other drivers tend to create an extra SGL and
10327     * refer to the TCE through that.
10328     */
10329     req = (pMpi2FWDownloadRequest)pt->request;
10330     tcsge = (pMpi2FWDownloadTCSGE_t)&req->SGL;
10331     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10332         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10333         mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10334     }
10335
10336     pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL) +
10337         sizeof(*tcsge);
10338     if (pt->request_size != pt->sgl_offset)
10339         NDBG15(("mpi_pre_fw_download(): Incorrect req size, "
10340             "0x%x, should be 0x%x, dataoutsz 0x%x",
10341             (int)pt->request_size, (int)pt->sgl_offset,
10342             (int)pt->dataout_size));
10343     if (pt->data_size < sizeof(MPI2_FW_DOWNLOAD_REPLY))
10344         NDBG15(("mpi_pre_fw_download(): Incorrect rep size, "
10345             "0x%x, should be 0x%x", pt->data_size,
10346             (int)sizeof(MPI2_FW_DOWNLOAD_REPLY)));
10347 }
10348
10349 /*
10350 * Prepare the pt for a SAS3 FW_DOWNLOAD request.
10351 */
10352 static void
10353 mpi_pre_fw_25_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10354 {
10355     pMpi2FWDownloadTCSGE_t tcsge;
10356     pMpi2FWDownloadRequest req2;
10357     pMpi25FWDownloadRequest req25;
10358
10359     /*
10360     * User requests should come in with the Transaction
10361     * context element where the SGL will go. The new firmware
10362     * Doesn't use TCE and has space in the main request for
10363     * this information. So move to the right place.
10364     */
10365     req2 = (pMpi2FWDownloadRequest)pt->request;
10366     req25 = (pMpi25FWDownloadRequest)pt->request;
10367     tcsge = (pMpi2FWDownloadTCSGE_t)&req2->SGL;
10368     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10369         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10370         mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10371     }
10372     req25->ImageOffset = tcsge->ImageOffset;
10373     req25->ImageSize = tcsge->ImageSize;
10374
10375     pt->sgl_offset = offsetof(MPI25_FW_DOWNLOAD_REQUEST, SGL);
10376     if (pt->request_size != pt->sgl_offset)
10377         NDBG15(("mpi_pre_fw_25_download(): Incorrect req size, "
10378             "0x%x, should be 0x%x, dataoutsz 0x%x",
10379             pt->request_size, pt->sgl_offset,
10380             pt->dataout_size));
10381     if (pt->data_size < sizeof(MPI25_FW_DOWNLOAD_REPLY))
10382         NDBG15(("mpi_pre_fw_25_download(): Incorrect rep size, "
10383             "0x%x, should be 0x%x", pt->data_size,
10384             (int)sizeof(MPI25_FW_UPLOAD_REPLY)));
10385 }
10386
10387 /*
10388 * Prepare the pt for a SAS2 FW_UPLOAD request.

```

```

10389 */
10390 static void
10391 mpi_pre_fw_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10392 {
10393     pMpi2FWUploadTCSGE_t tcsge;
10394     pMpi2FWUploadRequest_t req;
10395
10396     /*
10397      * If SAS3, call separate function.
10398      */
10399     if (mpt->m MPI25) {
10400         mpi_pre_fw_25_upload(mpt, pt);
10401         return;
10402     }
10403
10404     /*
10405      * User requests should come in with the Transaction
10406      * context element where the SGL will go. Putting the
10407      * SGL after that seems to work, but don't really know
10408      * why. Other drivers tend to create an extra SGL and
10409      * refer to the TCE through that.
10410      */
10411     req = (pMpi2FWUploadRequest_t)pt->request;
10412     tcsge = (pMpi2FWUploadTCSGE_t)&req->SGL;
10413     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10414         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10415         mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10416     }
10417
10418     pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL) +
10419         sizeof (*tcsge);
10420     if (pt->request_size != pt->sgl_offset)
10421         NDBG15(("mpi_pre_fw_upload(): Incorrect req size, "
10422             "0x%x, should be 0x%x, dataoutsz 0x%x",
10423             pt->request_size, pt->sgl_offset,
10424             pt->dataout_size));
10425     if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10426         NDBG15(("mpi_pre_fw_upload(): Incorrect rep size, "
10427             "0x%x, should be 0x%x", pt->data_size,
10428             (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10429 }
10430
10431 /*
10432  * Prepare the pt a SAS3 FW_UPLOAD request.
10433  */
10434 static void
10435 mpi_pre_fw_25_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10436 {
10437     pMpi2FWUploadTCSGE_t tcsge;
10438     pMpi2FWUploadRequest_t req2;
10439     pMpi25FWUploadRequest_t req25;
10440
10441     /*
10442      * User requests should come in with the Transaction
10443      * context element where the SGL will go. The new firmware
10444      * Doesn't use TCE and has space in the main request for
10445      * this information. So move to the right place.
10446      */
10447     req2 = (pMpi2FWUploadRequest_t)pt->request;
10448     req25 = (pMpi25FWUploadRequest_t)pt->request;
10449     tcsge = (pMpi2FWUploadTCSGE_t)&req2->SGL;
10450     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10451         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10452         mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10453     }
10454     req25->ImageOffset = tcsge->ImageOffset;

```

```

10455     req25->ImageSize = tcsge->ImageSize;
10456
10457     pt->sgl_offset = offsetof(MPI25_FW_UPLOAD_REQUEST, SGL);
10458     if (pt->request_size != pt->sgl_offset)
10459         NDBG15(("mpi_pre_fw_25_upload(): Incorrect req size, "
10460             "0x%x, should be 0x%x, dataoutsz 0x%x",
10461             pt->request_size, pt->sgl_offset,
10462             pt->dataout_size));
10463     if (pt->data_size < sizeof (MPI25_FW_UPLOAD_REPLY))
10464         NDBG15(("mpi_pre_fw_25_upload(): Incorrect rep size, "
10465             "0x%x, should be 0x%x", pt->data_size,
10466             (int)sizeof (MPI25_FW_UPLOAD_REPLY)));
10467 }
10468
10469 /*
10470  * Prepare the pt for an IOC_FACTS request.
10471  */
10472 static void
10473 mpi_pre_ioc_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10474 {
10475     #ifndef __lock_lint
10476         _NOTE(ARGUNUSED(mpt))
10477     #endif
10478     if (pt->request_size != sizeof (MPI2_IOC_FACTS_REQUEST))
10479         NDBG15(("mpi_pre_ioc_facts(): Incorrect req size, "
10480             "0x%x, should be 0x%x, dataoutsz 0x%x",
10481             pt->request_size,
10482             (int)sizeof (MPI2_IOC_FACTS_REQUEST),
10483             pt->dataout_size));
10484     if (pt->data_size != sizeof (MPI2_IOC_FACTS_REPLY))
10485         NDBG15(("mpi_pre_ioc_facts(): Incorrect rep size, "
10486             "0x%x, should be 0x%x", pt->data_size,
10487             (int)sizeof (MPI2_IOC_FACTS_REPLY)));
10488     pt->sgl_offset = (uint16_t)pt->request_size;
10489 }
10490
10491 /*
10492  * Prepare the pt for a PORT_FACTS request.
10493  */
10494 static void
10495 mpi_pre_port_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10496 {
10497     #ifndef __lock_lint
10498         _NOTE(ARGUNUSED(mpt))
10499     #endif
10500     if (pt->request_size != sizeof (MPI2_PORT_FACTS_REQUEST))
10501         NDBG15(("mpi_pre_port_facts(): Incorrect req size, "
10502             "0x%x, should be 0x%x, dataoutsz 0x%x",
10503             pt->request_size,
10504             (int)sizeof (MPI2_PORT_FACTS_REQUEST),
10505             pt->dataout_size));
10506     if (pt->data_size != sizeof (MPI2_PORT_FACTS_REPLY))
10507         NDBG15(("mpi_pre_port_facts(): Incorrect rep size, "
10508             "0x%x, should be 0x%x", pt->data_size,
10509             (int)sizeof (MPI2_PORT_FACTS_REPLY)));
10510     pt->sgl_offset = (uint16_t)pt->request_size;
10511 }
10512
10513 /*
10514  * Prepare pt for a SATA_PASSTHROUGH request.
10515  */
10516 static void
10517 mpi_pre_sata_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10518 {
10519     #ifndef __lock_lint
10520         _NOTE(ARGUNUSED(mpt))

```



```

10521 #endif
10522     pt->sgl_offset = offsetof(MPI2_SATA_PASSTHROUGH_REQUEST, SGL);
10523     if (pt->request_size != pt->sgl_offset)
10524         NDBG15(("mpi_pre_sata_passthrough(): Incorrect req size, "
10525             "0x%x, should be 0x%x, dataoutsz 0x%x",
10526             pt->request_size, pt->sgl_offset,
10527             pt->dataout_size));
10528     if (pt->data_size != sizeof (MPI2_SATA_PASSTHROUGH_REPLY))
10529         NDBG15(("mpi_pre_sata_passthrough(): Incorrect rep size, "
10530             "0x%x, should be 0x%x", pt->data_size,
10531             (int)sizeof (MPI2_SATA_PASSTHROUGH_REPLY)));
10532 }

10534 static void
10535 mpi_pre_smp_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10536 {
10537     #ifndef __lock_lint
10538         _NOTE(ARGUNUSED(mpt))
10539     #endif
10540     pt->sgl_offset = offsetof(MPI2_SMP_PASSTHROUGH_REQUEST, SGL);
10541     if (pt->request_size != pt->sgl_offset)
10542         NDBG15(("mpi_pre_smp_passthrough(): Incorrect req size, "
10543             "0x%x, should be 0x%x, dataoutsz 0x%x",
10544             pt->request_size, pt->sgl_offset,
10545             pt->dataout_size));
10546     if (pt->data_size != sizeof (MPI2_SMP_PASSTHROUGH_REPLY))
10547         NDBG15(("mpi_pre_smp_passthrough(): Incorrect rep size, "
10548             "0x%x, should be 0x%x", pt->data_size,
10549             (int)sizeof (MPI2_SMP_PASSTHROUGH_REPLY)));
10550 }

10552 /*
10553  * Prepare pt for a CONFIG request.
10554  */
10555 static void
10556 mpi_pre_config(mptsas_t *mpt, mptsas_pt_request_t *pt)
10557 {
10558     #ifndef __lock_lint
10559         _NOTE(ARGUNUSED(mpt))
10560     #endif
10561     pt->sgl_offset = offsetof(MPI2_CONFIG_REQUEST, PageBufferSGE);
10562     if (pt->request_size != pt->sgl_offset)
10563         NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10564             "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10565             pt->sgl_offset, pt->dataout_size));
10566     if (pt->data_size != sizeof (MPI2_CONFIG_REPLY))
10567         NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10568             "should be 0x%x", pt->data_size,
10569             (int)sizeof (MPI2_CONFIG_REPLY)));
10570     pt->simple = 1;
10571 }

10573 /*
10574  * Prepare pt for a SCSI_IO_REQ request.
10575  */
10576 static void
10577 mpi_pre_scsi_io_req(mptsas_t *mpt, mptsas_pt_request_t *pt)
10578 {
10579     #ifndef __lock_lint
10580         _NOTE(ARGUNUSED(mpt))
10581     #endif
10582     pt->sgl_offset = offsetof(MPI2_SCSI_IO_REQUEST, SGL);
10583     if (pt->request_size != pt->sgl_offset)
10584         NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10585             "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10586             pt->sgl_offset,

```

```

10587         pt->dataout_size));
10588     if (pt->data_size != sizeof (MPI2_SCSI_IO_REPLY))
10589         NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10590             "should be 0x%x", pt->data_size,
10591             (int)sizeof (MPI2_SCSI_IO_REPLY)));
10592 }

10594 /*
10595  * Prepare the mptsas_cmd for a SAS_IO_UNIT_CONTROL request.
10596  */
10597 static void
10598 mpi_pre_sas_io_unit_control(mptsas_t *mpt, mptsas_pt_request_t *pt)
10599 {
10600     #ifndef __lock_lint
10601         _NOTE(ARGUNUSED(mpt))
10602     #endif
10603     pt->sgl_offset = (uint16_t)pt->request_size;
10604 }

10606 /*
10607  * A set of functions to prepare an mptsas_cmd for the various
10608  * supported requests.
10609  */
10610 static struct mptsas_func {
10611     U8          Function;
10612     char        *Name;
10613     mptsas_pre_f *f_pre;
10614 } mptsas_func_list[] = {
10615     { MPI2_FUNCTION_IOC_FACTS, "IOC_FACTS",          mpi_pre_ioc_facts },
10616     { MPI2_FUNCTION_PORT_FACTS, "PORT_FACTS",        mpi_pre_port_facts },
10617     { MPI2_FUNCTION_FW_DOWNLOAD, "FW_DOWNLOAD",      mpi_pre_fw_download },
10618     { MPI2_FUNCTION_FW_UPLOAD, "FW_UPLOAD",          mpi_pre_fw_upload },
10619     { MPI2_FUNCTION_SATA_PASSTHROUGH, "SATA_PASSTHROUGH",
10620       mpi_pre_sata_passthrough },
10621     { MPI2_FUNCTION_SMP_PASSTHROUGH, "SMP_PASSTHROUGH",
10622       mpi_pre_smp_passthrough },
10623     { MPI2_FUNCTION_SCSI_IO_REQUEST, "SCSI_IO_REQUEST",
10624       mpi_pre_scsi_io_req },
10625     { MPI2_FUNCTION_CONFIG, "CONFIG",                mpi_pre_config },
10626     { MPI2_FUNCTION_SAS_IO_UNIT_CONTROL, "SAS_IO_UNIT_CONTROL",
10627       mpi_pre_sas_io_unit_control },
10628     { 0xFF, NULL, NULL } /* list end */
10629 };

10631 static void
10632 mptsas_prep_sgl_offset(mptsas_t *mpt, mptsas_pt_request_t *pt)
10633 {
10634     pMPI2RequestHeader_t  hdr;
10635     struct mptsas_func    *f;

10637     hdr = (pMPI2RequestHeader_t)pt->request;

10639     for (f = mptsas_func_list; f->f_pre != NULL; f++) {
10640         if (hdr->Function == f->Function) {
10641             f->f_pre(mpt, pt);
10642             NDBG15(("mptsas_prep_sgl_offset: Function %s",
10643                 "sgl_offset 0x%x", f->Name,
10644                 pt->sgl_offset));
10645             return;
10646         }
10647     }
10648     NDBG15(("mptsas_prep_sgl_offset: Unknown Function 0x%02x",
10649         "returning req_size 0x%x for sgl_offset",
10650         hdr->Function, pt->request_size));
10651     pt->sgl_offset = (uint16_t)pt->request_size;
10652 }

```

```

10655 static int
10656 mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
10657   uint8_t *data, uint32_t request_size, uint32_t reply_size,
10658   uint32_t data_size, uint32_t direction, uint8_t *dataout,
10659   uint32_t dataout_size, short timeout, int mode)
10660 {
10661     mptsas_pt_request_t      pt;
10662     mptsas_dma_alloc_state_t data_dma_state;
10663     mptsas_dma_alloc_state_t dataout_dma_state;
10664     caddr_t                  memp;
10665     mptsas_cmd_t             *cmd = NULL;
10666     struct scsi_pkt          *pkt;
10667     uint32_t                  reply_len = 0, sense_len = 0;
10668     pMPI2RequestHeader_t     request_hdrp;
10669     pMPI2RequestHeader_t     request_msg;
10670     pMPI2DefaultReply_t      reply_msg;
10671     Mpi2SCSIIOReply_t        rep_msg;
10672     int                       i, status = 0, pt_flags = 0, rv = 0;
10673     int                       rvalue;
10674     uint8_t                   function;
10675
10676     ASSERT(mutex_owned(&mpt->m_mutex));
10677
10678     reply_msg = (pMPI2DefaultReply_t)(&rep_msg);
10679     bzero(reply_msg, sizeof(MPI2_DEFAULT_REPLY));
10680     request_msg = kmem_zalloc(request_size, KM_SLEEP);
10681
10682     mutex_exit(&mpt->m_mutex);
10683     /*
10684      * copy in the request buffer since it could be used by
10685      * another thread when the pt request into waitq
10686      */
10687     if (ddi_copyin(request, request_msg, request_size, mode)) {
10688         mutex_enter(&mpt->m_mutex);
10689         status = EFAULT;
10690         mptsas_log(mpt, CE_WARN, "failed to copy request data");
10691         goto out;
10692     }
10693     mutex_enter(&mpt->m_mutex);
10694
10695     function = request_msg->Function;
10696     if (function == MPI2_FUNCTION_SCSI_TASK_MGMT) {
10697         pMpi2SCSITaskManagementRequest_t task;
10698         task = (pMpi2SCSITaskManagementRequest_t)request_msg;
10699         mptsas_setup_bus_reset_delay(mpt);
10700         rv = mptsas_ioc_task_management(mpt, task->TaskType,
10701           task->DevHandle, (int)task->LUN[1], reply, reply_size,
10702           mode);
10703
10704         if (rv != TRUE) {
10705             status = EIO;
10706             mptsas_log(mpt, CE_WARN, "task management failed");
10707         }
10708         goto out;
10709     }
10710
10711     if (data_size != 0) {
10712         data_dma_state.size = data_size;
10713         if (mptsas_dma_alloc(mpt, &data_dma_state) != DDI_SUCCESS) {
10714             status = ENOMEM;
10715             mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10716               "resource");
10717             goto out;
10718         }
10719     }

```

```

10719     pt_flags |= MPTSAS_DATA_ALLOCATED;
10720     if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10721         mutex_exit(&mpt->m_mutex);
10722         for (i = 0; i < data_size; i++) {
10723             if (ddi_copyin(data + i, (uint8_t *)
10724               data_dma_state.memp + i, 1, mode)) {
10725                 mutex_enter(&mpt->m_mutex);
10726                 status = EFAULT;
10727                 mptsas_log(mpt, CE_WARN, "failed to "
10728                   "copy read data");
10729                 goto out;
10730             }
10731         }
10732         mutex_enter(&mpt->m_mutex);
10733     }
10734 } else {
10735     bzero(&data_dma_state, sizeof(data_dma_state));
10736 }
10737
10738 if (dataout_size != 0) {
10739     dataout_dma_state.size = dataout_size;
10740     if (mptsas_dma_alloc(mpt, &dataout_dma_state) != DDI_SUCCESS) {
10741         status = ENOMEM;
10742         mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10743           "resource");
10744         goto out;
10745     }
10746     pt_flags |= MPTSAS_DATAOUT_ALLOCATED;
10747     mutex_exit(&mpt->m_mutex);
10748     for (i = 0; i < dataout_size; i++) {
10749         if (ddi_copyin(dataout + i, (uint8_t *)
10750           dataout_dma_state.memp + i, 1, mode)) {
10751             mutex_enter(&mpt->m_mutex);
10752             mptsas_log(mpt, CE_WARN, "failed to copy out "
10753               "data");
10754             status = EFAULT;
10755             goto out;
10756         }
10757     }
10758     mutex_enter(&mpt->m_mutex);
10759 } else {
10760     bzero(&dataout_dma_state, sizeof(dataout_dma_state));
10761 }
10762
10763 if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
10764     status = EAGAIN;
10765     mptsas_log(mpt, CE_NOTE, "event ack command pool is full");
10766     goto out;
10767 }
10768 pt_flags |= MPTSAS_REQUEST_POOL_CMD;
10769
10770 bzero((caddr_t)cmd, sizeof(*cmd));
10771 bzero((caddr_t)pkt, scsi_pkt_size());
10772 bzero((caddr_t)&pt, sizeof(pt));
10773
10774 cmd->ioc_cmd_slot = (uint32_t)(rvalue);
10775
10776 pt.request = (uint8_t *)request_msg;
10777 pt.direction = direction;
10778 pt.simple = 0;
10779 pt.request_size = request_size;
10780 pt.data_size = data_size;
10781 pt.dataout_size = dataout_size;
10782 pt.data_cookie = data_dma_state.cookie;
10783 pt.dataout_cookie = dataout_dma_state.cookie;
10784 mptsas_prep_sgl_offset(mpt, &pt);

```

```

10786     /*
10787     * Form a blank cmd/pkt to store the acknowledgement message
10788     */
10789     pkt->pkt_cdbp      = (opaque_t)&cmd->cmd_cdb[0];
10790     pkt->pkt_scbp      = (opaque_t)&cmd->cmd_scb;
10791     pkt->pkt_ha_private = (opaque_t)&pt;
10792     pkt->pkt_flags     = FLAG_HEAD;
10793     pkt->pkt_time      = timeout;
10794     cmd->cmd_pkt       = pkt;
10795     cmd->cmd_flags     = CFLAG_CMDIOC | CFLAG_PASSTHRU;

10797     /*
10798     * Save the command in a slot
10799     */
10800     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
10801         /*
10802         * Once passthru command get slot, set cmd_flags
10803         * CFLAG_PREPARED.
10804         */
10805         cmd->cmd_flags |= CFLAG_PREPARED;
10806         mptsas_start_passthru(mpt, cmd);
10807     } else {
10808         mptsas_waitq_add(mpt, cmd);
10809     }

10811     while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
10812         cv_wait(&mpt->m_passthru_cv, &mpt->m_mutex);
10813     }

10815     if (cmd->cmd_flags & CFLAG_PREPARED) {
10816         memp = mpt->m_req_frame + (mpt->m_req_frame_size *
10817             cmd->cmd_slot);
10818         request_hdrp = (pMPI2RequestHeader_t)memp;
10819     }

10821     if (cmd->cmd_flags & CFLAG_TIMEOUT) {
10822         status = ETIMEDOUT;
10823         mptsas_log(mpt, CE_WARN, "passthrough command timeout");
10824         pt_flags |= MPTSAS_CMD_TIMEOUT;
10825         goto out;
10826     }

10828     if (cmd->cmd_rfm) {
10829         /*
10830         * cmd_rfm is zero means the command reply is a CONTEXT
10831         * reply and no PCI Write to post the free reply SMFA
10832         * because no reply message frame is used.
10833         * cmd_rfm is non-zero means the reply is a ADDRESS
10834         * reply and reply message frame is used.
10835         */
10836         pt_flags |= MPTSAS_ADDRESS_REPLY;
10837         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
10838             DDI_DMA_SYNC_FORCPU);
10839         reply_msg = (pMPI2DefaultReply_t)
10840             (mpt->m_reply_frame + (cmd->cmd_rfm -
10841                 mpt->m_reply_frame_dma_addr));
10842     }

10844     mptsas_fma_check(mpt, cmd);
10845     if (pkt->pkt_reason == CMD_TRAN_ERR) {
10846         status = EAGAIN;
10847         mptsas_log(mpt, CE_WARN, "passthru fma error");
10848         goto out;
10849     }
10850     if (pkt->pkt_reason == CMD_RESET) {

```

```

10851         status = EAGAIN;
10852         mptsas_log(mpt, CE_WARN, "ioc reset abort passthru");
10853         goto out;
10854     }

10856     if (pkt->pkt_reason == CMD_INCOMPLETE) {
10857         status = EIO;
10858         mptsas_log(mpt, CE_WARN, "passthrough command incomplete");
10859         goto out;
10860     }

10862     mutex_exit(&mpt->m_mutex);
10863     if (cmd->cmd_flags & CFLAG_PREPARED) {
10864         function = request_hdrp->Function;
10865         if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10866             (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10867             reply_len = sizeof (MPI2_SCSI_IO_REPLY);
10868             sense_len = reply_size - reply_len;
10869         } else {
10870             reply_len = reply_size;
10871             sense_len = 0;
10872         }

10874         for (i = 0; i < reply_len; i++) {
10875             if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
10876                 mode)) {
10877                 mutex_enter(&mpt->m_mutex);
10878                 status = EFAULT;
10879                 mptsas_log(mpt, CE_WARN, "failed to copy out "
10880                     "reply data");
10881                 goto out;
10882             }
10883         }
10884         for (i = 0; i < sense_len; i++) {
10885             if (ddi_copyout((uint8_t *)request_hdrp + 64 + i,
10886                 reply + reply_len + i, 1, mode)) {
10887                 mutex_enter(&mpt->m_mutex);
10888                 status = EFAULT;
10889                 mptsas_log(mpt, CE_WARN, "failed to copy out "
10890                     "sense data");
10891                 goto out;
10892             }
10893         }
10894     }

10896     if (data_size) {
10897         if (direction != MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10898             (void) ddi_dma_sync(data_dma_state.handle, 0, 0,
10899                 DDI_DMA_SYNC_FORCPU);
10900             for (i = 0; i < data_size; i++) {
10901                 if (ddi_copyout((uint8_t *)
10902                     data_dma_state.memp + i), data + i, 1,
10903                     mode)) {
10904                     mutex_enter(&mpt->m_mutex);
10905                     status = EFAULT;
10906                     mptsas_log(mpt, CE_WARN, "failed to "
10907                         "copy out the reply data");
10908                     goto out;
10909                 }
10910             }
10911         }
10912     }
10913     mutex_enter(&mpt->m_mutex);
10914 out:
10915     /*
10916     * Put the reply frame back on the free queue, increment the free

```

```

10917  * index, and write the new index to the free index register. But only
10918  * if this reply is an ADDRESS reply.
10919  */
10920  if (pt_flags & MPTSAS_ADDRESS_REPLY) {
10921      ddi_put32(mpt->m_acc_free_queue_hdl,
10922              &((uint32_t *) (void *) mpt->m_free_queue)[mpt->m_free_index],
10923              cmd->cmd_rfm);
10924      (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
10925                        DDI_DMA_SYNC_FORDEV);
10926      if (++mpt->m_free_index == mpt->m_free_queue_depth) {
10927          mpt->m_free_index = 0;
10928      }
10929      ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
10930              mpt->m_free_index);
10931  }
10932  if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
10933      mptsas_remove_cmd(mpt, cmd);
10934      pt_flags &= (~MPTSAS_REQUEST_POOL_CMD);
10935  }
10936  if (pt_flags & MPTSAS_REQUEST_POOL_CMD)
10937      mptsas_return_to_pool(mpt, cmd);
10938  if (pt_flags & MPTSAS_DATA_ALLOCATED) {
10939      if (mptsas_check_dma_handle(data_dma_state.handle) !=
10940          DDI_SUCCESS) {
10941          ddi_fm_service_impact(mpt->m_dip,
10942                              DDI_SERVICE_UNAFFECTED);
10943          status = EFAULT;
10944      }
10945      mptsas_dma_free(&data_dma_state);
10946  }
10947  if (pt_flags & MPTSAS_DATAOUT_ALLOCATED) {
10948      if (mptsas_check_dma_handle(dataout_dma_state.handle) !=
10949          DDI_SUCCESS) {
10950          ddi_fm_service_impact(mpt->m_dip,
10951                              DDI_SERVICE_UNAFFECTED);
10952          status = EFAULT;
10953      }
10954      mptsas_dma_free(&dataout_dma_state);
10955  }
10956  if (pt_flags & MPTSAS_CMD_TIMEOUT) {
10957      if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
10958          mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
10959      }
10960  }
10961  if (request_msg)
10962      kmem_free(request_msg, request_size);

10964  return (status);
10965  }

10967  static int
10968  mptsas_pass_thru(mptsas_t *mpt, mptsas_pass_thru_t *data, int mode)
10969  {
10970      /*
10971       * If timeout is 0, set timeout to default of 60 seconds.
10972       */
10973      if (data->Timeout == 0) {
10974          data->Timeout = MPTSAS_PASS_THRU_TIME_DEFAULT;
10975      }

10977      if (((data->DataSize == 0) &&
10978          (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_NONE)) ||
10979          ((data->DataSize != 0) &&
10980          ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_READ) ||
10981          (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_WRITE) ||
10982          (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) &&

```

```

10983      (data->DataOutSize != 0)))) {
10984          if (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) {
10985              data->DataDirection = MPTSAS_PASS_THRU_DIRECTION_READ;
10986          } else {
10987              data->DataOutSize = 0;
10988          }
10989          /*
10990           * Send passthru request messages
10991           */
10992          return (mptsas_do_passthru(mpt,
10993                                   (uint8_t *) ((uintptr_t) data->PtrRequest),
10994                                   (uint8_t *) ((uintptr_t) data->PtrReply),
10995                                   (uint8_t *) ((uintptr_t) data->PtrData),
10996                                   data->RequestSize, data->ReplySize,
10997                                   data->DataSize, data->DataDirection,
10998                                   (uint8_t *) ((uintptr_t) data->PtrDataOut),
10999                                   data->DataOutSize, data->Timeout, mode));
11000      } else {
11001          return (EINVAL);
11002      }
11003  }

11005  static uint8_t
11006  mptsas_get_fw_diag_buffer_number(mptsas_t *mpt, uint32_t unique_id)
11007  {
11008      uint8_t index;

11010      for (index = 0; index < MPI2_DIAG_BUF_TYPE_COUNT; index++) {
11011          if (mpt->m_fw_diag_buffer_list[index].unique_id == unique_id) {
11012              return (index);
11013          }
11014      }

11016      return (MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND);
11017  }

11019  static void
11020  mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
11021  {
11022      pMpi2DiagBufferPostRequest_t  pDiag_post_msg;
11023      pMpi2DiagReleaseRequest_t     pDiag_release_msg;
11024      struct scsi_pkt                *pkt = cmd->cmd_pkt;
11025      mptsas_diag_request_t          *diag = pkt->pkt_ha_private;
11026      uint32_t                        request_desc_low, i;

11028      ASSERT(mutex_owned(&mpt->m_mutex));

11030      /*
11031       * Form the diag message depending on the post or release function.
11032       */
11033      if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
11034          pDiag_post_msg = (pMpi2DiagBufferPostRequest_t)
11035              (mpt->m_req_frame + (mpt->m_req_frame_size *
11036                cmd->cmd_slot));
11037          bzero(pDiag_post_msg, mpt->m_req_frame_size);
11038          ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
11039                diag->function);
11040          ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
11041                diag->pBuffer->buffer_type);
11042          ddi_put8(mpt->m_acc_req_frame_hdl,
11043                &pDiag_post_msg->ExtendedType,
11044                diag->pBuffer->extended_type);
11045          ddi_put32(mpt->m_acc_req_frame_hdl,
11046                &pDiag_post_msg->BufferLength,
11047                diag->pBuffer->buffer_data.size);
11048          for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);

```

```

11049         i++) {
11050             ddi_put32(mpt->m_acc_req_frame_hdl,
11051                 &pDiag_post_msg->ProductSpecific[i],
11052                 diag->pBuffer->product_specific[i]);
11053         }
11054         ddi_put32(mpt->m_acc_req_frame_hdl,
11055             &pDiag_post_msg->BufferAddress.Low,
11056             (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11057                 & 0xffffffff));
11058         ddi_put32(mpt->m_acc_req_frame_hdl,
11059             &pDiag_post_msg->BufferAddress.High,
11060             (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11061                 >> 32));
11062     } else {
11063         pDiag_release_msg = (pMpi2DiagReleaseRequest_t)
11064             (mpt->m_req_frame + (mpt->m_req_frame_size *
11065                 cmd->cmd_slot));
11066         bzero(pDiag_release_msg, mpt->m_req_frame_size);
11067         ddi_put8(mpt->m_acc_req_frame_hdl,
11068             &pDiag_release_msg->Function, diag->function);
11069         ddi_put8(mpt->m_acc_req_frame_hdl,
11070             &pDiag_release_msg->BufferType,
11071             diag->pBuffer->buffer_type);
11072     }
11073
11074     /*
11075     * Send the message
11076     */
11077     (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
11078         DDI_DMA_SYNC_FORDEV);
11079     request_desc_low = (cmd->cmd_slot << 16) +
11080         MPI2_REQ_DESCRIPTOR_FLAGS_DEFAULT_TYPE;
11081     cmd->cmd_rfm = NULL;
11082     MPTSAS_START_CMD(mpt, request_desc_low, 0);
11083     if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
11084         DDI_SUCCESS) ||
11085         (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
11086         DDI_SUCCESS)) {
11087         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11088     }
11089 }
11090
11091 static int
11092 mptsas_post_fw_diag_buffer(mptsas_t *mpt,
11093     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code)
11094 {
11095     mptsas_diag_request_t    diag;
11096     int                      status, slot_num, post_flags = 0;
11097     mptsas_cmd_t            *cmd = NULL;
11098     struct scsi_pkt         *pkt;
11099     pMpi2DiagBufferPostReply_t reply;
11100     uint16_t                iocstatus;
11101     uint32_t                iocloginfo, transfer_length;
11102
11103     /*
11104     * If buffer is not enabled, just leave.
11105     */
11106     *return_code = MPTSAS_FW_DIAG_ERROR_POST_FAILED;
11107     if (!pBuffer->enabled) {
11108         status = DDI_FAILURE;
11109         goto out;
11110     }
11111
11112     /*
11113     * Clear some flags initially.
11114     */

```

```

11115         pBuffer->force_release = FALSE;
11116         pBuffer->valid_data = FALSE;
11117         pBuffer->owned_by_firmware = FALSE;
11118
11119     /*
11120     * Get a cmd buffer from the cmd buffer pool
11121     */
11122     if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11123         status = DDI_FAILURE;
11124         mptsas_log(mpt, CE_NOTE, "command pool is full: Post FW Diag");
11125         goto out;
11126     }
11127     post_flags |= MPTSAS_REQUEST_POOL_CMD;
11128
11129     bzero((caddr_t)cmd, sizeof (*cmd));
11130     bzero((caddr_t)pkt, scsi_pkt_size());
11131
11132     cmd->ioc_cmd_slot = (uint32_t)(slot_num);
11133
11134     diag.pBuffer = pBuffer;
11135     diag.function = MPI2_FUNCTION_DIAG_BUFFER_POST;
11136
11137     /*
11138     * Form a blank cmd/pkt to store the acknowledgement message
11139     */
11140     pkt->pkt_ha_private = (opaque_t)&diag;
11141     pkt->pkt_flags = FLAG_HEAD;
11142     pkt->pkt_time = 60;
11143     cmd->cmd_pkt = pkt;
11144     cmd->cmd_flags = CFLAG_CMDIOC | CFLAG_FW_DIAG;
11145
11146     /*
11147     * Save the command in a slot
11148     */
11149     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11150         /*
11151         * Once passthru command get slot, set cmd_flags
11152         * CFLAG_PREPARED.
11153         */
11154         cmd->cmd_flags |= CFLAG_PREPARED;
11155         mptsas_start_diag(mpt, cmd);
11156     } else {
11157         mptsas_waitq_add(mpt, cmd);
11158     }
11159
11160     while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11161         cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11162     }
11163
11164     if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11165         status = DDI_FAILURE;
11166         mptsas_log(mpt, CE_WARN, "Post FW Diag command timeout");
11167         goto out;
11168     }
11169
11170     /*
11171     * cmd_rfm points to the reply message if a reply was given. Check the
11172     * IOCStatus to make sure everything went OK with the FW diag request
11173     * and set buffer flags.
11174     */
11175     if (cmd->cmd_rfm) {
11176         post_flags |= MPTSAS_ADDRESS_REPLY;
11177         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11178             DDI_DMA_SYNC_FORCPU);
11179         reply = (pMpi2DiagBufferPostReply_t)(mpt->m_reply_frame +
11180             (cmd->cmd_rfm - mpt->m_reply_frame_dma_addr));

```

```

11182     /*
11183     * Get the reply message data
11184     */
11185     iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11186         &reply->IOCStatus);
11187     iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11188         &reply->IOCLogInfo);
11189     transfer_length = ddi_get32(mpt->m_acc_reply_frame_hdl,
11190         &reply->TransferLength);
11191
11192     /*
11193     * If post failed quit.
11194     */
11195     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
11196         status = DDI_FAILURE;
11197         NDBG13(("post FW Diag Buffer failed: IOCStatus=0x%x, "
11198             "IOCLogInfo=0x%x, TransferLength=0x%x", iocstatus,
11199             iocloginfo, transfer_length));
11200         goto out;
11201     }
11202
11203     /*
11204     * Post was successful.
11205     */
11206     pBuffer->valid_data = TRUE;
11207     pBuffer->owned_by_firmware = TRUE;
11208     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11209     status = DDI_SUCCESS;
11210 }
11211
11212 out:
11213 /*
11214 * Put the reply frame back on the free queue, increment the free
11215 * index, and write the new index to the free index register. But only
11216 * if this reply is an ADDRESS reply.
11217 */
11218 if (post_flags & MPTSAS_ADDRESS_REPLY) {
11219     ddi_put32(mpt->m_acc_free_queue_hdl,
11220         &((uint32_t *)mpt->m_free_queue)[mpt->m_free_index],
11221         cmd->cmd_rfm);
11222     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11223         DDI_DMA_SYNC_FORDEV);
11224     if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11225         mpt->m_free_index = 0;
11226     }
11227     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11228         mpt->m_free_index);
11229 }
11230 if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11231     mptsas_remove_cmd(mpt, cmd);
11232     post_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11233 }
11234 if (post_flags & MPTSAS_REQUEST_POOL_CMD) {
11235     mptsas_return_to_pool(mpt, cmd);
11236 }
11237
11238 return (status);
11239 }
11240
11241 static int
11242 mptsas_release_fw_diag_buffer(mptsas_t *mpt,
11243     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
11244     uint32_t diag_type)
11245 {
11246     mptsas_diag_request_t diag;

```

```

11247     int                status, slot_num, rel_flags = 0;
11248     mptsas_cmd_t       *cmd = NULL;
11249     struct scsi_pkt    *pkt;
11250     pMpi2DiagReleaseReply_t reply;
11251     uint16_t           iocstatus;
11252     uint32_t           iocloginfo;
11253
11254     /*
11255     * If buffer is not enabled, just leave.
11256     */
11257     *return_code = MPTSAS_FW_DIAG_ERROR_RELEASE_FAILED;
11258     if (!pBuffer->enabled) {
11259         mptsas_log(mpt, CE_NOTE, "This buffer type is not supported "
11260             "by the IOC");
11261         status = DDI_FAILURE;
11262         goto out;
11263     }
11264
11265     /*
11266     * Clear some flags initially.
11267     */
11268     pBuffer->force_release = FALSE;
11269     pBuffer->valid_data = FALSE;
11270     pBuffer->owned_by_firmware = FALSE;
11271
11272     /*
11273     * Get a cmd buffer from the cmd buffer pool
11274     */
11275     if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11276         status = DDI_FAILURE;
11277         mptsas_log(mpt, CE_NOTE, "command pool is full: Release FW "
11278             "Diag");
11279         goto out;
11280     }
11281     rel_flags |= MPTSAS_REQUEST_POOL_CMD;
11282
11283     bzero((caddr_t)cmd, sizeof (*cmd));
11284     bzero((caddr_t)pkt, scsi_pkt_size());
11285
11286     cmd->ioc_cmd_slot = (uint32_t)(slot_num);
11287
11288     diag.pBuffer = pBuffer;
11289     diag.function = MPI2_FUNCTION_DIAG_RELEASE;
11290
11291     /*
11292     * Form a blank cmd/pkt to store the acknowledgement message
11293     */
11294     pkt->pkt_ha_private = (opaque_t)&diag;
11295     pkt->pkt_flags = FLAG_HEAD;
11296     pkt->pkt_time = 60;
11297     cmd->cmd_pkt = pkt;
11298     cmd->cmd_flags = CFLAG_CMDIOC | CFLAG_FW_DIAG;
11299
11300     /*
11301     * Save the command in a slot
11302     */
11303     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11304         /*
11305         * Once passthru command get slot, set cmd_flags
11306         * CFLAG_PREPARED.
11307         */
11308         cmd->cmd_flags |= CFLAG_PREPARED;
11309         mptsas_start_diag(mpt, cmd);
11310     } else {
11311         mptsas_waitq_add(mpt, cmd);
11312     }

```

```

11314 while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11315     cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11316 }

11318 if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11319     status = DDI_FAILURE;
11320     mptsas_log(mpt, CE_WARN, "Release FW Diag command timeout");
11321     goto out;
11322 }

11324 /*
11325  * cmd_rfm points to the reply message if a reply was given. Check the
11326  * IOCStatus to make sure everything went OK with the FW diag request
11327  * and set buffer flags.
11328  */
11329 if (cmd->cmd_rfm) {
11330     rel_flags |= MPTSAS_ADDRESS_REPLY;
11331     (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11332         DDI_DMA_SYNC_FORCPU);
11333     reply = (pMpi2DiagReleaseReply_t)(mpt->m_reply_frame +
11334         (cmd->cmd_rfm - mpt->m_reply_frame_dma_addr));

11336     /*
11337      * Get the reply message data
11338      */
11339     iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11340         &reply->IOCStatus);
11341     iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11342         &reply->IOCLogInfo);

11344     /*
11345      * If release failed quit.
11346      */
11347     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) ||
11348         pBuffer->owned_by_firmware) {
11349         status = DDI_FAILURE;
11350         NDBG13(("release FW Diag Buffer failed: "
11351             "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
11352             iocloginfo));
11353         goto out;
11354     }

11356     /*
11357      * Release was successful.
11358      */
11359     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11360     status = DDI_SUCCESS;

11362     /*
11363      * If this was for an UNREGISTER diag type command, clear the
11364      * unique ID.
11365      */
11366     if (diag_type == MPTSAS_FW_DIAG_TYPE_UNREGISTER) {
11367         pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11368     }
11369 }

11371 out:
11372 /*
11373  * Put the reply frame back on the free queue, increment the free
11374  * index, and write the new index to the free index register. But only
11375  * if this reply is an ADDRESS reply.
11376  */
11377 if (rel_flags & MPTSAS_ADDRESS_REPLY) {
11378     ddi_put32(mpt->m_acc_free_queue_hdl,

```

```

11379         &((uint32_t *) (void *) mpt->m_free_queue)[mpt->m_free_index],
11380         cmd->cmd_rfm);
11381     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11382         DDI_DMA_SYNC_FORDEV);
11383     if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11384         mpt->m_free_index = 0;
11385     }
11386     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11387         mpt->m_free_index);
11388 }
11389 if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11390     mptsas_remove_cmd(mpt, cmd);
11391     rel_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11392 }
11393 if (rel_flags & MPTSAS_REQUEST_POOL_CMD) {
11394     mptsas_return_to_pool(mpt, cmd);
11395 }

11397 return (status);
11398 }

11400 static int
11401 mptsas_diag_register(mptsas_t *mpt, mptsas_fw_diag_register_t *diag_register,
11402     uint32_t *return_code)
11403 {
11404     mptsas_fw_diagnostic_buffer_t *pBuffer;
11405     uint8_t extended_type, buffer_type, i;
11406     uint32_t buffer_size;
11407     uint32_t unique_id;
11408     int status;

11410     ASSERT(mutex_owned(&mpt->m_mutex));

11412     extended_type = diag_register->ExtendedType;
11413     buffer_type = diag_register->BufferType;
11414     buffer_size = diag_register->RequestedBufferSize;
11415     unique_id = diag_register->UniqueID;

11417     /*
11418      * Check for valid buffer type
11419      */
11420     if (buffer_type >= MPI2_DIAG_BUF_TYPE_COUNT) {
11421         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11422         return (DDI_FAILURE);
11423     }

11425     /*
11426      * Get the current buffer and look up the unique ID. The unique ID
11427      * should not be found. If it is, the ID is already in use.
11428      */
11429     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11430     pBuffer = &mpt->m_fw_diag_buffer_list[buffer_type];
11431     if (i != MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11432         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11433         return (DDI_FAILURE);
11434     }

11436     /*
11437      * The buffer's unique ID should not be registered yet, and the given
11438      * unique ID cannot be 0.
11439      */
11440     if ((pBuffer->unique_id != MPTSAS_FW_DIAG_INVALID_UID) ||
11441         (unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11442         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11443         return (DDI_FAILURE);
11444     }

```

```

11446  /*
11447  * If this buffer is already posted as immediate, just change owner.
11448  */
11449  if (pBuffer->immediate && pBuffer->owned_by_firmware &&
11450      (pBuffer->unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11451      pBuffer->immediate = FALSE;
11452      pBuffer->unique_id = unique_id;
11453      return (DDI_SUCCESS);
11454  }
11455
11456  /*
11457  * Post a new buffer after checking if it's enabled. The DMA buffer
11458  * that is allocated will be contiguous (sgl_len = 1).
11459  */
11460  if (!pBuffer->enabled) {
11461      *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11462      return (DDI_FAILURE);
11463  }
11464  bzero(&pBuffer->buffer_data, sizeof (mptsas_dma_alloc_state_t));
11465  pBuffer->buffer_data.size = buffer_size;
11466  if (mptsas_dma_alloc(mpt, &pBuffer->buffer_data) != DDI_SUCCESS) {
11467      mptsas_log(mpt, CE_WARN, "failed to alloc DMA resource for "
11468                "diag buffer: size = %d bytes", buffer_size);
11469      *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11470      return (DDI_FAILURE);
11471  }
11472
11473  /*
11474  * Copy the given info to the diag buffer and post the buffer.
11475  */
11476  pBuffer->buffer_type = buffer_type;
11477  pBuffer->immediate = FALSE;
11478  if (buffer_type == MPI2_DIAG_BUF_TYPE_TRACE) {
11479      for (i = 0; i < (sizeof (pBuffer->product_specific) / 4);
11480          i++) {
11481          pBuffer->product_specific[i] =
11482              diag_register->ProductSpecific[i];
11483      }
11484  }
11485  pBuffer->extended_type = extended_type;
11486  pBuffer->unique_id = unique_id;
11487  status = mptsas_post_fw_diag_buffer(mpt, pBuffer, return_code);
11488
11489  if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11490      DDI_SUCCESS) {
11491      mptsas_log(mpt, CE_WARN, "Check of DMA handle failed in "
11492                "mptsas_diag_register.");
11493      ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11494      status = DDI_FAILURE;
11495  }
11496
11497  /*
11498  * In case there was a failure, free the DMA buffer.
11499  */
11500  if (status == DDI_FAILURE) {
11501      mptsas_dma_free(&pBuffer->buffer_data);
11502  }
11503
11504  return (status);
11505 }
11506
11507 static int
11508 mptsas_diag_unregister(mptsas_t *mpt,
11509     mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code)
11510 {

```

```

11511     mptsas_fw_diagnostic_buffer_t *pBuffer;
11512     uint8_t i;
11513     uint32_t unique_id;
11514     int status;
11515
11516     ASSERT(mutex_owned(&mpt->m_mutex));
11517
11518     unique_id = diag_unregister->UniqueId;
11519
11520     /*
11521     * Get the current buffer and look up the unique ID. The unique ID
11522     * should be there.
11523     */
11524     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11525     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11526         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11527         return (DDI_FAILURE);
11528     }
11529
11530     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11531
11532     /*
11533     * Try to release the buffer from FW before freeing it. If release
11534     * fails, don't free the DMA buffer in case FW tries to access it
11535     * later. If buffer is not owned by firmware, can't release it.
11536     */
11537     if (!pBuffer->owned_by_firmware) {
11538         status = DDI_SUCCESS;
11539     } else {
11540         status = mptsas_release_fw_diag_buffer(mpt, pBuffer,
11541             return_code, MPTSAS_FW_DIAG_TYPE_UNREGISTER);
11542     }
11543
11544     /*
11545     * At this point, return the current status no matter what happens with
11546     * the DMA buffer.
11547     */
11548     pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11549     if (status == DDI_SUCCESS) {
11550         if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11551             DDI_SUCCESS) {
11552             mptsas_log(mpt, CE_WARN, "Check of DMA handle failed "
11553                       "in mptsas_diag_unregister.");
11554             ddi_fm_service_impact(mpt->m_dip,
11555                 DDI_SERVICE_UNAFFECTED);
11556         }
11557         mptsas_dma_free(&pBuffer->buffer_data);
11558     }
11559
11560     return (status);
11561 }
11562
11563 static int
11564 mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
11565     uint32_t *return_code)
11566 {
11567     mptsas_fw_diagnostic_buffer_t *pBuffer;
11568     uint8_t i;
11569     uint32_t unique_id;
11570
11571     ASSERT(mutex_owned(&mpt->m_mutex));
11572
11573     unique_id = diag_query->UniqueId;
11574
11575     /*
11576     * If ID is valid, query on ID.

```



```

11577     * If ID is invalid, query on buffer type.
11578     */
11579     if (unique_id == MPTSAS_FW_DIAG_INVALID_UID) {
11580         i = diag_query->BufferType;
11581         if (i >= MPI2_DIAG_BUF_TYPE_COUNT) {
11582             *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11583             return (DDI_FAILURE);
11584         }
11585     } else {
11586         i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11587         if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11588             *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11589             return (DDI_FAILURE);
11590         }
11591     }
11592
11593     /*
11594     * Fill query structure with the diag buffer info.
11595     */
11596     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11597     diag_query->BufferType = pBuffer->buffer_type;
11598     diag_query->ExtendedType = pBuffer->extended_type;
11599     if (diag_query->BufferType == MPI2_DIAG_BUF_TYPE_TRACE) {
11600         for (i = 0; i < (sizeof (diag_query->ProductSpecific) / 4);
11601             i++) {
11602             diag_query->ProductSpecific[i] =
11603                 pBuffer->product_specific[i];
11604         }
11605     }
11606     diag_query->TotalBufferSize = pBuffer->buffer_data.size;
11607     diag_query->DriverAddedBufferSize = 0;
11608     diag_query->UniqueId = pBuffer->unique_id;
11609     diag_query->ApplicationFlags = 0;
11610     diag_query->DiagnosticFlags = 0;
11611
11612     /*
11613     * Set/Clear application flags
11614     */
11615     if (pBuffer->immediate) {
11616         diag_query->ApplicationFlags &= ~MPTSAS_FW_DIAG_FLAG_APP_OWNED;
11617     } else {
11618         diag_query->ApplicationFlags |= MPTSAS_FW_DIAG_FLAG_APP_OWNED;
11619     }
11620     if (pBuffer->valid_data || pBuffer->owned_by_firmware) {
11621         diag_query->ApplicationFlags |=
11622             MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11623     } else {
11624         diag_query->ApplicationFlags &=
11625             ~MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11626     }
11627     if (pBuffer->owned_by_firmware) {
11628         diag_query->ApplicationFlags |=
11629             MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11630     } else {
11631         diag_query->ApplicationFlags &=
11632             ~MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11633     }
11634
11635     return (DDI_SUCCESS);
11636 }
11637
11638 static int
11639 mptsas_diag_read_buffer(mptsas_t *mpt,
11640     mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
11641     uint32_t *return_code, int ioctl_mode)
11642 {

```

```

11643     mptsas_fw_diagnostic_buffer_t *pBuffer;
11644     uint8_t i, *pData;
11645     uint32_t unique_id, byte;
11646     int status;
11647
11648     ASSERT(mutex_owned(&mpt->m_mutex));
11649
11650     unique_id = diag_read_buffer->UniqueId;
11651
11652     /*
11653     * Get the current buffer and look up the unique ID. The unique ID
11654     * should be there.
11655     */
11656     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11657     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11658         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11659         return (DDI_FAILURE);
11660     }
11661
11662     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11663
11664     /*
11665     * Make sure requested read is within limits
11666     */
11667     if (diag_read_buffer->StartingOffset + diag_read_buffer->BytesToRead >
11668         pBuffer->buffer_data.size) {
11669         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11670         return (DDI_FAILURE);
11671     }
11672
11673     /*
11674     * Copy the requested data from DMA to the diag_read_buffer. The DMA
11675     * buffer that was allocated is one contiguous buffer.
11676     */
11677     pData = (uint8_t *) (pBuffer->buffer_data.memp +
11678         diag_read_buffer->StartingOffset);
11679     (void) ddi_dma_sync(pBuffer->buffer_data.handle, 0, 0,
11680         DDI_DMA_SYNC_FORCPU);
11681     for (byte = 0; byte < diag_read_buffer->BytesToRead; byte++) {
11682         if (ddi_copyout(pData + byte, ioctl_buf + byte, 1, ioctl_mode)
11683             != 0) {
11684             return (DDI_FAILURE);
11685         }
11686     }
11687     diag_read_buffer->Status = 0;
11688
11689     /*
11690     * Set or clear the Force Release flag.
11691     */
11692     if (pBuffer->force_release) {
11693         diag_read_buffer->Flags |= MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11694     } else {
11695         diag_read_buffer->Flags &= ~MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11696     }
11697
11698     /*
11699     * If buffer is to be reregistered, make sure it's not already owned by
11700     * firmware first.
11701     */
11702     status = DDI_SUCCESS;
11703     if (!pBuffer->owned_by_firmware) {
11704         if (diag_read_buffer->Flags & MPTSAS_FW_DIAG_FLAG_REREGISTER) {
11705             status = mptsas_post_fw_diag_buffer(mpt, pBuffer,
11706                 return_code);
11707         }
11708     }

```

```

11710     return (status);
11711 }

11713 static int
11714 mptsas_diag_release(mptsas_t *mpt, mptsas_fw_diag_release_t *diag_release,
11715     uint32_t *return_code)
11716 {
11717     mptsas_fw_diagnostic_buffer_t *pBuffer;
11718     uint8_t i;
11719     uint32_t unique_id;
11720     int status;

11722     ASSERT(mutex_owned(&mpt->m_mutex));

11724     unique_id = diag_release->UniqueId;

11726     /*
11727     * Get the current buffer and look up the unique ID.  The unique ID
11728     * should be there.
11729     */
11730     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11731     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11732         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11733         return (DDI_FAILURE);
11734     }

11736     pBuffer = &mpt->m_fw_diag_buffer_list[i];

11738     /*
11739     * If buffer is not owned by firmware, it's already been released.
11740     */
11741     if (!pBuffer->owned_by_firmware) {
11742         *return_code = MPTSAS_FW_DIAG_ERROR_ALREADY_RELEASED;
11743         return (DDI_FAILURE);
11744     }

11746     /*
11747     * Release the buffer.
11748     */
11749     status = mptsas_release_fw_diag_buffer(mpt, pBuffer, return_code,
11750         MPTSAS_FW_DIAG_TYPE_RELEASE);
11751     return (status);
11752 }

11754 static int
11755 mptsas_do_diag_action(mptsas_t *mpt, uint32_t action, uint8_t *diag_action,
11756     uint32_t length, uint32_t *return_code, int ioctl_mode)
11757 {
11758     mptsas_fw_diag_register_t diag_register;
11759     mptsas_fw_diag_unregister_t diag_unregister;
11760     mptsas_fw_diag_query_t diag_query;
11761     mptsas_diag_read_buffer_t diag_read_buffer;
11762     mptsas_fw_diag_release_t diag_release;
11763     int status = DDI_SUCCESS;
11764     uint32_t original_return_code, read_buf_len;

11766     ASSERT(mutex_owned(&mpt->m_mutex));

11768     original_return_code = *return_code;
11769     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;

11771     switch (action) {
11772     case MPTSAS_FW_DIAG_TYPE_REGISTER:
11773         if (!length) {
11774             *return_code =

```

```

11775         MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11776         status = DDI_FAILURE;
11777         break;
11778     }
11779     if (ddi_copyin(diag_action, &diag_register,
11780         sizeof (diag_register), ioctl_mode) != 0) {
11781         return (DDI_FAILURE);
11782     }
11783     status = mptsas_diag_register(mpt, &diag_register,
11784         return_code);
11785     break;

11787 case MPTSAS_FW_DIAG_TYPE_UNREGISTER:
11788     if (length < sizeof (diag_unregister)) {
11789         *return_code =
11790             MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11791         status = DDI_FAILURE;
11792         break;
11793     }
11794     if (ddi_copyin(diag_action, &diag_unregister,
11795         sizeof (diag_unregister), ioctl_mode) != 0) {
11796         return (DDI_FAILURE);
11797     }
11798     status = mptsas_diag_unregister(mpt, &diag_unregister,
11799         return_code);
11800     break;

11802 case MPTSAS_FW_DIAG_TYPE_QUERY:
11803     if (length < sizeof (diag_query)) {
11804         *return_code =
11805             MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11806         status = DDI_FAILURE;
11807         break;
11808     }
11809     if (ddi_copyin(diag_action, &diag_query,
11810         sizeof (diag_query), ioctl_mode) != 0) {
11811         return (DDI_FAILURE);
11812     }
11813     status = mptsas_diag_query(mpt, &diag_query,
11814         return_code);
11815     if (status == DDI_SUCCESS) {
11816         if (ddi_copyout(&diag_query, diag_action,
11817             sizeof (diag_query), ioctl_mode) != 0) {
11818             return (DDI_FAILURE);
11819         }
11820     }
11821     break;

11823 case MPTSAS_FW_DIAG_TYPE_READ_BUFFER:
11824     if (ddi_copyin(diag_action, &diag_read_buffer,
11825         sizeof (diag_read_buffer) - 4, ioctl_mode) != 0) {
11826         return (DDI_FAILURE);
11827     }
11828     read_buf_len = sizeof (diag_read_buffer) -
11829         sizeof (diag_read_buffer.DataBuffer) +
11830         diag_read_buffer.BytesToRead;
11831     if (length < read_buf_len) {
11832         *return_code =
11833             MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11834         status = DDI_FAILURE;
11835         break;
11836     }
11837     status = mptsas_diag_read_buffer(mpt,
11838         &diag_read_buffer, diag_action +
11839         sizeof (diag_read_buffer) - 4, return_code,
11840         ioctl_mode);

```

```

11841         if (status == DDI_SUCCESS) {
11842             if (ddi_copyout(&diag_read_buffer, diag_action,
11843                 sizeof (diag_read_buffer) - 4, ioctl_mode)
11844                 != 0) {
11845                 return (DDI_FAILURE);
11846             }
11847         }
11848         break;
11850     case MPTSAS_FW_DIAG_TYPE_RELEASE:
11851         if (length < sizeof (diag_release)) {
11852             *return_code =
11853                 MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11854             status = DDI_FAILURE;
11855             break;
11856         }
11857         if (ddi_copyin(diag_action, &diag_release,
11858             sizeof (diag_release), ioctl_mode) != 0) {
11859             return (DDI_FAILURE);
11860         }
11861         status = mptsas_diag_release(mpt, &diag_release,
11862             return_code);
11863         break;
11865     default:
11866         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11867         status = DDI_FAILURE;
11868         break;
11869 }
11871 if ((status == DDI_FAILURE) &&
11872     (original_return_code == MPTSAS_FW_DIAG_NEW) &&
11873     (*return_code != MPTSAS_FW_DIAG_ERROR_SUCCESS)) {
11874     status = DDI_SUCCESS;
11875 }
11877 return (status);
11878 }
11880 static int
11881 mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *user_data, int mode)
11882 {
11883     int                status;
11884     mptsas_diag_action_t  driver_data;
11886     ASSERT(mutex_owned(&mpt->m_mutex));
11888     /*
11889     * Copy the user data to a driver data buffer.
11890     */
11891     if (ddi_copyin(user_data, &driver_data, sizeof (mptsas_diag_action_t),
11892         mode) == 0) {
11893         /*
11894         * Send diag action request if Action is valid
11895         */
11896         if (driver_data.Action == MPTSAS_FW_DIAG_TYPE_REGISTER ||
11897             driver_data.Action == MPTSAS_FW_DIAG_TYPE_UNREGISTER ||
11898             driver_data.Action == MPTSAS_FW_DIAG_TYPE_QUERY ||
11899             driver_data.Action == MPTSAS_FW_DIAG_TYPE_READ_BUFFER ||
11900             driver_data.Action == MPTSAS_FW_DIAG_TYPE_RELEASE) {
11901             status = mptsas_do_diag_action(mpt, driver_data.Action,
11902                 (void *) (uintptr_t) driver_data.PtrDiagAction,
11903                 driver_data.Length, &driver_data.ReturnCode,
11904                 mode);
11905             if (status == DDI_SUCCESS) {
11906                 if (ddi_copyout(&driver_data.ReturnCode,

```

```

11907             &user_data->ReturnCode,
11908             sizeof (user_data->ReturnCode), mode)
11909             != 0) {
11910                 status = EFAULT;
11911             } else {
11912                 status = 0;
11913             }
11914         } else {
11915             status = EIO;
11916         }
11917     } else {
11918         status = EINVAL;
11919     }
11920 } else {
11921     status = EFAULT;
11922 }
11924 return (status);
11925 }
11927 /*
11928 * This routine handles the "event query" ioctl.
11929 */
11930 static int
11931 mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data, int mode,
11932     int *rval)
11933 {
11934     int                status;
11935     mptsas_event_query_t  driverdata;
11936     uint8_t            i;
11938     driverdata.Entries = MPTSAS_EVENT_QUEUE_SIZE;
11940     mutex_enter(&mpt->m_mutex);
11941     for (i = 0; i < 4; i++) {
11942         driverdata.Types[i] = mpt->m_event_mask[i];
11943     }
11944     mutex_exit(&mpt->m_mutex);
11946     if (ddi_copyout(&driverdata, data, sizeof (driverdata), mode) != 0) {
11947         status = EFAULT;
11948     } else {
11949         *rval = MPTIOCTL_STATUS_GOOD;
11950         status = 0;
11951     }
11953     return (status);
11954 }
11956 /*
11957 * This routine handles the "event enable" ioctl.
11958 */
11959 static int
11960 mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data, int mode,
11961     int *rval)
11962 {
11963     int                status;
11964     mptsas_event_enable_t  driverdata;
11965     uint8_t            i;
11967     if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
11968         mutex_enter(&mpt->m_mutex);
11969         for (i = 0; i < 4; i++) {
11970             mpt->m_event_mask[i] = driverdata.Types[i];
11971         }
11972         mutex_exit(&mpt->m_mutex);

```

```

11974         *rval = MPTIOCTL_STATUS_GOOD;
11975         status = 0;
11976     } else {
11977         status = EFAULT;
11978     }
11979     return (status);
11980 }

11982 /*
11983  * This routine handles the "event report" ioctl.
11984  */
11985 static int
11986 mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data, int mode,
11987                    int *rval)
11988 {
11989     int                status;
11990     mptsas_event_report_t  driverdata;

11992     mutex_enter(&mpt->m_mutex);

11994     if (ddi_copyin(&data->Size, &driverdata.Size, sizeof (driverdata.Size),
11995                  mode) == 0) {
11996         if (driverdata.Size >= sizeof (mpt->m_events)) {
11997             if (ddi_copyout(mpt->m_events, data->Events,
11998                           sizeof (mpt->m_events), mode) != 0) {
11999                 status = EFAULT;
12000             } else {
12001                 if (driverdata.Size > sizeof (mpt->m_events)) {
12002                     driverdata.Size =
12003                         sizeof (mpt->m_events);
12004                     if (ddi_copyout(&driverdata.Size,
12005                                   &data->Size,
12006                                   sizeof (driverdata.Size),
12007                                   mode) != 0) {
12008                         status = EFAULT;
12009                     } else {
12010                         *rval = MPTIOCTL_STATUS_GOOD;
12011                         status = 0;
12012                     }
12013                 } else {
12014                     *rval = MPTIOCTL_STATUS_GOOD;
12015                     status = 0;
12016                 }
12017             }
12018         } else {
12019             *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12020             status = 0;
12021         }
12022     } else {
12023         status = EFAULT;
12024     }

12026     mutex_exit(&mpt->m_mutex);
12027     return (status);
12028 }

12030 static void
12031 mptsas_lookup_pci_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12032 {
12033     int                *reg_data;
12034     uint_t              reglen;

12036     /*
12037      * Lookup the 'reg' property and extract the other data
12038      */

```

```

12039     if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12040                                  DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
12041         DDI_PROP_SUCCESS) {
12042         /*
12043          * Extract the PCI data from the 'reg' property first DWORD.
12044          * The entry looks like the following:
12045          * First DWORD:
12046          * Bits 0 - 7 8-bit Register number
12047          * Bits 8 - 10 3-bit Function number
12048          * Bits 11 - 15 5-bit Device number
12049          * Bits 16 - 23 8-bit Bus number
12050          * Bits 24 - 25 2-bit Address Space type identifier
12051          */
12052         adapter_data->PciInformation.u.bits.BusNumber =
12053             (reg_data[0] & 0x00FF0000) >> 16;
12054         adapter_data->PciInformation.u.bits.DeviceNumber =
12055             (reg_data[0] & 0x0000F800) >> 11;
12056         adapter_data->PciInformation.u.bits.FunctionNumber =
12057             (reg_data[0] & 0x00000700) >> 8;
12058         ddi_prop_free((void *)reg_data);
12059     } else {
12060         /*
12061          * If we can't determine the PCI data then we fill in FF's for
12062          * the data to indicate this.
12063          */
12064         adapter_data->PCIDeviceHwId = 0xFFFFFFFF;
12065         adapter_data->MpiPortNumber = 0xFFFFFFFF;
12066         adapter_data->PciInformation.u.AsDWORD = 0xFFFFFFFF;
12067     }

12070     /*
12071      * Saved in the mpt->m_fwversion
12072      */
12073     adapter_data->MpiFirmwareVersion = mpt->m_fwversion;
12074 }

12076 static void
12077 mptsas_read_adapter_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12078 {
12079     char                *driver_verstr = MPTSAS_MOD_STRING;

12081     mptsas_lookup_pci_data(mpt, adapter_data);
12082     adapter_data->AdapterType = mpt->m_mpi25 ?
12083         MPTIOCTL_ADAPTER_TYPE_SAS3 :
12084         MPTIOCTL_ADAPTER_TYPE_SAS2;
12085     adapter_data->PCIDeviceHwId = (uint32_t)mpt->m_devid;
12086     adapter_data->PCIDeviceHwRev = (uint32_t)mpt->m_revid;
12087     adapter_data->SubsystemId = (uint32_t)mpt->m_ssid;
12088     adapter_data->SubsystemVendorId = (uint32_t)mpt->m_svid;
12089     (void) strcpy((char *)&adapter_data->DriverVersion[0], driver_verstr);
12090     adapter_data->BiosVersion = 0;
12091     (void) mptsas_get_bios_page3(mpt, &adapter_data->BiosVersion);
12092 }

12094 static void
12095 mptsas_read_pci_info(mptsas_t *mpt, mptsas_pci_info_t *pci_info)
12096 {
12097     int                *reg_data, i;
12098     uint_t              reglen;

12100     /*
12101      * Lookup the 'reg' property and extract the other data
12102      */
12103     if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12104                                  DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==

```

```

12105         DDI_PROP_SUCCESS) {
12106             /*
12107              * Extract the PCI data from the 'reg' property first DWORD.
12108              * The entry looks like the following:
12109              * First DWORD:
12110              * Bits 8 - 10 3-bit Function number
12111              * Bits 11 - 15 5-bit Device number
12112              * Bits 16 - 23 8-bit Bus number
12113              */
12114             pci_info->BusNumber = (reg_data[0] & 0x00FF0000) >> 16;
12115             pci_info->DeviceNumber = (reg_data[0] & 0x0000F800) >> 11;
12116             pci_info->FunctionNumber = (reg_data[0] & 0x00000700) >> 8;
12117             ddi_prop_free((void *)reg_data);
12118         } else {
12119             /*
12120              * If we can't determine the PCI info then we fill in FF's for
12121              * the data to indicate this.
12122              */
12123             pci_info->BusNumber = 0xFFFFFFFF;
12124             pci_info->DeviceNumber = 0xFF;
12125             pci_info->FunctionNumber = 0xFF;
12126         }
12127
12128         /*
12129          * Now get the interrupt vector and the pci header. The vector can
12130          * only be 0 right now. The header is the first 256 bytes of config
12131          * space.
12132          */
12133         pci_info->InterruptVector = 0;
12134         for (i = 0; i < sizeof (pci_info->PciHeader); i++) {
12135             pci_info->PciHeader[i] = pci_config_get8(mpt->m_config_handle,
12136             i);
12137         }
12138     }
12139
12140     static int
12141     mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data, int mode)
12142     {
12143         int status = 0;
12144         mptsas_reg_access_t driverdata;
12145
12146         mutex_enter(&mpt->m_mutex);
12147         if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
12148             switch (driverdata.Command) {
12149                 /*
12150                  * IO access is not supported.
12151                  */
12152                 case REG_IO_READ:
12153                 case REG_IO_WRITE:
12154                     mptsas_log(mpt, CE_WARN, "IO access is not "
12155                     "supported. Use memory access.");
12156                     status = EINVAL;
12157                     break;
12158
12159                 case REG_MEM_READ:
12160                     driverdata.RegData = ddi_get32(mpt->m_datap,
12161                     (uint32_t *) (void *) mpt->m_reg +
12162                     driverdata.RegOffset);
12163                     if (ddi_copyout(&driverdata.RegData,
12164                     &data->RegData,
12165                     sizeof (driverdata.RegData), mode) != 0) {
12166                         mptsas_log(mpt, CE_WARN, "Register "
12167                         "Read Failed");
12168                         status = EFAULT;
12169                     }
12170                     break;

```

```

12172             case REG_MEM_WRITE:
12173                 ddi_put32(mpt->m_datap,
12174                 (uint32_t *) (void *) mpt->m_reg +
12175                 driverdata.RegOffset,
12176                 driverdata.RegData);
12177                 break;
12178
12179             default:
12180                 status = EINVAL;
12181                 break;
12182         }
12183     } else {
12184         status = EFAULT;
12185     }
12186
12187     mutex_exit(&mpt->m_mutex);
12188     return (status);
12189 }
12190
12191 static int
12192 led_control(mptsas_t *mpt, intptr_t data, int mode)
12193 {
12194     int ret = 0;
12195     mptsas_led_control_t lc;
12196     mptsas_target_t *ptgt;
12197
12198     if (ddi_copyin((void *) data, &lc, sizeof (lc), mode) != 0) {
12199         return (EFAULT);
12200     }
12201
12202     if ((lc.Command != MPTSAS_LEDCTL_FLAG_SET &&
12203         lc.Command != MPTSAS_LEDCTL_FLAG_GET) ||
12204         lc.Led < MPTSAS_LEDCTL_LED_MIN ||
12205         lc.Led > MPTSAS_LEDCTL_LED_MAX ||
12206         (lc.Command == MPTSAS_LEDCTL_FLAG_SET && lc.LedStatus != 0 &&
12207         lc.LedStatus != 1)) {
12208         return (EINVAL);
12209     }
12210
12211     if ((lc.Command == MPTSAS_LEDCTL_FLAG_SET && (mode & FWRITE) == 0) ||
12212         (lc.Command == MPTSAS_LEDCTL_FLAG_GET && (mode & FREAD) == 0))
12213         return (EACCESS);
12214
12215     /* Locate the target we're interrogating... */
12216     mutex_enter(&mpt->m_mutex);
12217     ptgt = rehash_linear_search(mpt->m_targets,
12218     mptsas_target_eval_slot, &lc);
12219     if (ptgt == NULL) {
12220         /* We could not find a target for that enclosure/slot. */
12221         mutex_exit(&mpt->m_mutex);
12222         return (ENOENT);
12223     }
12224
12225     if (lc.Command == MPTSAS_LEDCTL_FLAG_SET) {
12226         /* Update our internal LED state. */
12227         ptgt->m_led_status &= ~(1 << (lc.Led - 1));
12228         ptgt->m_led_status |= lc.LedStatus << (lc.Led - 1);
12229
12230         /* Flush it to the controller. */
12231         ret = mptsas_flush_led_status(mpt, ptgt);
12232         mutex_exit(&mpt->m_mutex);
12233         return (ret);
12234     }
12235
12236     /* Return our internal LED state. */

```

```

12237     lc.LedStatus = (ptgt->m_led_status >> (lc.Led - 1)) & 1;
12238     mutex_exit(&mpt->m_mutex);

12240     if (ddi_copyout(&lc, (void *)data, sizeof (lc), mode) != 0) {
12241         return (EFAULT);
12242     }

12244     return (0);
12245 }

12247 static int
12248 get_disk_info(mptsas_t *mpt, intptr_t data, int mode)
12249 {
12250     uint16_t i = 0;
12251     uint16_t count = 0;
12252     int ret = 0;
12253     mptsas_target_t *ptgt;
12254     mptsas_disk_info_t *di;
12255     STRUCT_DECL(mptsas_get_disk_info, gdi);

12257     if ((mode & FREAD) == 0)
12258         return (EACCES);

12260     STRUCT_INIT(gdi, get_udatamodel());

12262     if (ddi_copyin((void *)data, STRUCT_BUF(gdi), STRUCT_SIZE(gdi),
12263 mode) != 0) {
12264         return (EFAULT);
12265     }

12267     /* Find out how many targets there are. */
12268     mutex_enter(&mpt->m_mutex);
12269     for (ptgt = reflash_first(mpt->m_targets); ptgt != NULL;
12270 ptgt = reflash_next(mpt->m_targets, ptgt)) {
12271         count++;
12272     }
12273     mutex_exit(&mpt->m_mutex);

12275     /*
12276      * If we haven't been asked to copy out information on each target,
12277      * then just return the count.
12278      */
12279     STRUCT_FSET(gdi, DiskCount, count);
12280     if (STRUCT_FGETP(gdi, PtrDiskInfoArray) == NULL)
12281         goto copy_out;

12283     /*
12284      * If we haven't been given a large enough buffer to copy out into,
12285      * let the caller know.
12286      */
12287     if (STRUCT_FGET(gdi, DiskInfoArraySize) <
12288 count * sizeof (mptsas_disk_info_t)) {
12289         ret = ENOSPC;
12290         goto copy_out;
12291     }

12293     di = kmem_zalloc(count * sizeof (mptsas_disk_info_t), KM_SLEEP);

12295     mutex_enter(&mpt->m_mutex);
12296     for (ptgt = reflash_first(mpt->m_targets); ptgt != NULL;
12297 ptgt = reflash_next(mpt->m_targets, ptgt)) {
12298         if (i >= count) {
12299             /*
12300              * The number of targets changed while we weren't
12301              * looking, so give up.
12302              */

```

```

12303         reflash_rele(mpt->m_targets, ptgt);
12304         mutex_exit(&mpt->m_mutex);
12305         kmem_free(di, count * sizeof (mptsas_disk_info_t));
12306         return (EAGAIN);
12307     }
12308     di[i].Instance = mpt->m_instance;
12309     di[i].Enclosure = ptgt->m_enclosure;
12310     di[i].Slot = ptgt->m_slot_num;
12311     di[i].SasAddress = ptgt->m_addr.mta_wwn;
12312     i++;
12313 }
12314     mutex_exit(&mpt->m_mutex);
12315     STRUCT_FSET(gdi, DiskCount, i);

12317     /* Copy out the disk information to the caller. */
12318     if (ddi_copyout((void *)di, STRUCT_FGETP(gdi, PtrDiskInfoArray),
12319 i * sizeof (mptsas_disk_info_t), mode) != 0) {
12320         ret = EFAULT;
12321     }

12323     kmem_free(di, count * sizeof (mptsas_disk_info_t));

12325 copy_out:
12326     if (ddi_copyout(STRUCT_BUF(gdi), (void *)data, STRUCT_SIZE(gdi),
12327 mode) != 0) {
12328         ret = EFAULT;
12329     }

12331     return (ret);
12332 }

12334 static int
12335 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
12336 int *rval)
12337 {
12338     int
12339     mptsas_t
12340     mptsas_update_flash_t
12341     mptsas_pass_thru_t
12342     mptsas_adapter_data_t
12343     mptsas_pci_info_t
12344     int
12345     mptsas_target_t
12346     int
12347     dev_info_t
12348     mptsas_phymask_t
12349     struct devctl_iocdata
12350     char
12351     mptsas_target_t
12352     status = 0;
12353     *mpt;
12354     flashdata;
12355     passthru_data;
12356     adapter_data;
12357     pci_info;
12358     copylen;
12359     iport_flag = 0;
12360     *dip = NULL;
12361     phymask = 0;
12362     *dcp = NULL;
12363     *addr = NULL;
12364     *ptgt = NULL;

12353     *rval = MPTIOCTL_STATUS_GOOD;
12354     if (secpolicy_sys_config(credp, B_FALSE) != 0) {
12355         return (EPERM);
12356     }

12358     mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
12359     if (mpt == NULL) {
12360         /*
12361          * Called from iport node, get the states
12362          */
12363         iport_flag = 1;
12364         dip = mptsas_get_dip_from_dev(dev, &phymask);
12365         if (dip == NULL) {
12366             return (ENXIO);
12367         }
12368         mpt = DIP2MPT(dip);

```

```

12369     }
12370     /* Make sure power level is D0 before accessing registers */
12371     mutex_enter(&mpt->m_mutex);
12372     if (mpt->m_options & MPTSAS_OPT_PM) {
12373         (void) pm_busy_component(mpt->m_dip, 0);
12374         if (mpt->m_power_level != PM_LEVEL_D0) {
12375             mutex_exit(&mpt->m_mutex);
12376             if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
12377                 DDI_SUCCESS) {
12378                 mptsas_log(mpt, CE_WARN,
12379                     "mptsas%d: mptsas_ioctl: Raise power "
12380                     "request failed.", mpt->m_instance);
12381                 (void) pm_idle_component(mpt->m_dip, 0);
12382                 return (ENXIO);
12383             }
12384         } else {
12385             mutex_exit(&mpt->m_mutex);
12386         }
12387     } else {
12388         mutex_exit(&mpt->m_mutex);
12389     }
12391     if (iport_flag) {
12392         status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
12393         if (status != 0) {
12394             goto out;
12395         }
12396         /*
12397          * The following code control the OK2RM LED, it doesn't affect
12398          * the ioctl return status.
12399          */
12400         if ((cmd == DEVCTL_DEVICE_ONLINE) ||
12401             (cmd == DEVCTL_DEVICE_OFFLINE)) {
12402             if (ndi_dc_allochdl((void *)data, &dcp) !=
12403                 NDI_SUCCESS) {
12404                 goto out;
12405             }
12406             addr = ndi_dc_getaddr(dcp);
12407             ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
12408             if (ptgt == NULL) {
12409                 NDBG14(("mptsas_ioctl led control: tgt %s not "
12410                     "found", addr));
12411                 ndi_dc_freehdl(dcp);
12412                 goto out;
12413             }
12414             mutex_enter(&mpt->m_mutex);
12415             if (cmd == DEVCTL_DEVICE_ONLINE) {
12416                 ptgt->m_tgt_unconfigured = 0;
12417             } else if (cmd == DEVCTL_DEVICE_OFFLINE) {
12418                 ptgt->m_tgt_unconfigured = 1;
12419             }
12420             if (cmd == DEVCTL_DEVICE_OFFLINE) {
12421                 ptgt->m_led_status |=
12422                     (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12423             } else {
12424                 ptgt->m_led_status &=
12425                     ~(1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12426             }
12427             (void) mptsas_flush_led_status(mpt, ptgt);
12428             mutex_exit(&mpt->m_mutex);
12429             ndi_dc_freehdl(dcp);
12430         }
12431         goto out;
12432     }
12433     switch (cmd) {
12434         case MPTIOCTL_GET_DISK_INFO:

```

```

12435         status = get_disk_info(mpt, data, mode);
12436         break;
12437     case MPTIOCTL_LED_CONTROL:
12438         status = led_control(mpt, data, mode);
12439         break;
12440     case MPTIOCTL_UPDATE_FLASH:
12441         if (ddi_copyin((void *)data, &flashdata,
12442             sizeof (struct mptsas_update_flash), mode)) {
12443             status = EFAULT;
12444             break;
12445         }
12447         mutex_enter(&mpt->m_mutex);
12448         if (mptsas_update_flash(mpt,
12449             (caddr_t)(long)flashdata.PtrBuffer,
12450             flashdata.ImageSize, flashdata.ImageType, mode)) {
12451             status = EFAULT;
12452         }
12454         /*
12455          * Reset the chip to start using the new
12456          * firmware. Reset if failed also.
12457          */
12458         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12459         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
12460             status = EFAULT;
12461         }
12462         mutex_exit(&mpt->m_mutex);
12463         break;
12464     case MPTIOCTL_PASS_THRU:
12465         /*
12466          * The user has requested to pass through a command to
12467          * be executed by the MPT firmware. Call our routine
12468          * which does this. Only allow one passthru IOCTL at
12469          * one time. Other threads will block on
12470          * m_passthru_mutex, which is of adaptive variant.
12471          */
12472         if (ddi_copyin((void *)data, &passthru_data,
12473             sizeof (mptsas_pass_thru_t), mode)) {
12474             status = EFAULT;
12475             break;
12476         }
12477         mutex_enter(&mpt->m_passthru_mutex);
12478         mutex_enter(&mpt->m_mutex);
12479         status = mptsas_pass_thru(mpt, &passthru_data, mode);
12480         mutex_exit(&mpt->m_mutex);
12481         mutex_exit(&mpt->m_passthru_mutex);
12483         break;
12484     case MPTIOCTL_GET_ADAPTER_DATA:
12485         /*
12486          * The user has requested to read adapter data. Call
12487          * our routine which does this.
12488          */
12489         bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
12490         if (ddi_copyin((void *)data, (void *)&adapter_data,
12491             sizeof (mptsas_adapter_data_t), mode)) {
12492             status = EFAULT;
12493             break;
12494         }
12495         if (adapter_data.StructureLength >=
12496             sizeof (mptsas_adapter_data_t)) {
12497             adapter_data.StructureLength = (uint32_t)
12498                 sizeof (mptsas_adapter_data_t);
12499             copylen = sizeof (mptsas_adapter_data_t);
12500             mutex_enter(&mpt->m_mutex);

```

```

12501     mptsas_read_adapter_data(mpt, &adapter_data);
12502     mutex_exit(&mpt->m_mutex);
12503     } else {
12504         adapter_data.StructureLength = (uint32_t)
12505             sizeof (mptsas_adapter_data_t);
12506         copylen = sizeof (adapter_data.StructureLength);
12507         *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12508     }
12509     if (ddi_copyout((void *)(&adapter_data), (void *)data,
12510         copylen, mode) != 0) {
12511         status = EFAULT;
12512     }
12513     break;
12514 case MPTIOCTL_GET_PCI_INFO:
12515     /*
12516      * The user has requested to read pci info. Call
12517      * our routine which does this.
12518      */
12519     bzero(&pci_info, sizeof (mptsas_pci_info_t));
12520     mutex_enter(&mpt->m_mutex);
12521     mptsas_read_pci_info(mpt, &pci_info);
12522     mutex_exit(&mpt->m_mutex);
12523     if (ddi_copyout((void *)(&pci_info), (void *)data,
12524         sizeof (mptsas_pci_info_t), mode) != 0) {
12525         status = EFAULT;
12526     }
12527     break;
12528 case MPTIOCTL_RESET_ADAPTER:
12529     mutex_enter(&mpt->m_mutex);
12530     mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12531     if ((mptsas_restart_ioc(mpt) == DDI_FAILURE) {
12532         mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
12533             "failed");
12534         status = EFAULT;
12535     }
12536     mutex_exit(&mpt->m_mutex);
12537     break;
12538 case MPTIOCTL_DIAG_ACTION:
12539     /*
12540      * The user has done a diag buffer action. Call our
12541      * routine which does this. Only allow one diag action
12542      * at one time.
12543      */
12544     mutex_enter(&mpt->m_mutex);
12545     if (mpt->m_diag_action_in_progress) {
12546         mutex_exit(&mpt->m_mutex);
12547         return (EBUSY);
12548     }
12549     mpt->m_diag_action_in_progress = 1;
12550     status = mptsas_diag_action(mpt,
12551         (mptsas_diag_action_t *)data, mode);
12552     mpt->m_diag_action_in_progress = 0;
12553     mutex_exit(&mpt->m_mutex);
12554     break;
12555 case MPTIOCTL_EVENT_QUERY:
12556     /*
12557      * The user has done an event query. Call our routine
12558      * which does this.
12559      */
12560     status = mptsas_event_query(mpt,
12561         (mptsas_event_query_t *)data, mode, rval);
12562     break;
12563 case MPTIOCTL_EVENT_ENABLE:
12564     /*
12565      * The user has done an event enable. Call our routine
12566      * which does this.

```

```

12567     /*
12568     status = mptsas_event_enable(mpt,
12569         (mptsas_event_enable_t *)data, mode, rval);
12570     break;
12571 case MPTIOCTL_EVENT_REPORT:
12572     /*
12573      * The user has done an event report. Call our routine
12574      * which does this.
12575      */
12576     status = mptsas_event_report(mpt,
12577         (mptsas_event_report_t *)data, mode, rval);
12578     break;
12579 case MPTIOCTL_REG_ACCESS:
12580     /*
12581      * The user has requested register access. Call our
12582      * routine which does this.
12583      */
12584     status = mptsas_reg_access(mpt,
12585         (mptsas_reg_access_t *)data, mode);
12586     break;
12587 default:
12588     status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
12589         rval);
12590     break;
12591     }
12592 }
12593 out:
12594     return (status);
12595 }
12597 int
12598 mptsas_restart_ioc(mptsas_t *mpt)
12599 {
12600     int             rval = DDI_SUCCESS;
12601     mptsas_target_t *ptgt = NULL;
12602
12603     ASSERT(mutex_owned(&mpt->m_mutex));
12604
12605     /*
12606      * Set a flag telling I/O path that we're processing a reset. This is
12607      * needed because after the reset is complete, the hash table still
12608      * needs to be rebuilt. If I/Os are started before the hash table is
12609      * rebuilt, I/O errors will occur. This flag allows I/Os to be marked
12610      * so that they can be retried.
12611      */
12612     mpt->m_in_reset = TRUE;
12613
12614     /*
12615      * Set all throttles to HOLD
12616      */
12617     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
12618         ptgt = rehash_next(mpt->m_targets, ptgt)) {
12619         mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
12620     }
12621
12622     /*
12623      * Disable interrupts
12624      */
12625     MPTSAS_DISABLE_INTR(mpt);
12626
12627     /*
12628      * Abort all commands: outstanding commands, commands in waitq and
12629      * tx_waitq.
12630      */
12631     mptsas_flush_hba(mpt);

```



```

12633  /*
12634  * Reinitialize the chip.
12635  */
12636  if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
12637      rval = DDI_FAILURE;
12638  }
12640  /*
12641  * Enable interrupts again
12642  */
12643  MPTSAS_ENABLE_INTR(mpt);
12645  /*
12646  * If mptsas_init_chip was successful, update the driver data.
12647  */
12648  if (rval == DDI_SUCCESS) {
12649      mptsas_update_driver_data(mpt);
12650  }
12652  /*
12653  * Reset the throttles
12654  */
12655  for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
12656      ptgt = rehash_next(mpt->m_targets, ptgt)) {
12657      mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
12658  }
12660  mptsas_doneq_empty(mpt);
12661  mptsas_restart_hba(mpt);
12663  if (rval != DDI_SUCCESS) {
12664      mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
12665      ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
12666  }
12668  /*
12669  * Clear the reset flag so that I/Os can continue.
12670  */
12671  mpt->m_in_reset = FALSE;
12673  return (rval);
12674  }
12676  static int
12677  mptsas_init_chip(mptsas_t *mpt, int first_time)
12678  {
12679      ddi_dma_cookie_t    cookie;
12680      uint32_t            i;
12681      int                 rval;
12683  /*
12684  * Check to see if the firmware image is valid
12685  */
12686  if (ddi_get32(mpt->m_datap, &mpt->m_reg->HostDiagnostic) &
12687      MPI2_DIAG_FLASH_BAD_SIG) {
12688      mptsas_log(mpt, CE_WARN, "mptsas bad flash signature!");
12689      goto fail;
12690  }
12692  /*
12693  * Reset the chip
12694  */
12695  rval = mptsas_ioc_reset(mpt, first_time);
12696  if (rval == MPTSAS_RESET_FAIL) {
12697      mptsas_log(mpt, CE_WARN, "hard reset failed!");
12698      goto fail;

```

```

12699  }
12701  if ((rval == MPTSAS_SUCCESS_MUR) && (!first_time)) {
12702      goto mur;
12703  }
12704  /*
12705  * Setup configuration space
12706  */
12707  if (mptsas_config_space_init(mpt) == FALSE) {
12708      mptsas_log(mpt, CE_WARN, "mptsas_config_space_init "
12709          "failed!");
12710      goto fail;
12711  }
12713  /*
12714  * IOC facts can change after a diag reset so all buffers that are
12715  * based on these numbers must be de-allocated and re-allocated. Get
12716  * new IOC facts each time chip is initialized.
12717  */
12718  if (mptsas_ioc_get_facts(mpt) == DDI_FAILURE) {
12719      mptsas_log(mpt, CE_WARN, "mptsas_ioc_get_facts failed");
12720      goto fail;
12721  }
12723  mpt->m_targets = rehash_create(MPTSAS_TARGET_BUCKET_COUNT,
12724      mptsas_target_addr_hash, mptsas_target_addr_cmp,
12725      mptsas_target_free, sizeof (mptsas_target_t),
12726      offsetof(mptsas_target_t, m_link),
12727      offsetof(mptsas_target_t, m_addr), KM_SLEEP);
12729  if (mptsas_alloc_active_slots(mpt, KM_SLEEP)) {
12730      goto fail;
12731  }
12732  /*
12733  * Allocate request message frames, reply free queue, reply descriptor
12734  * post queue, and reply message frames using latest IOC facts.
12735  */
12736  if (mptsas_alloc_request_frames(mpt) == DDI_FAILURE) {
12737      mptsas_log(mpt, CE_WARN, "mptsas_alloc_request_frames failed");
12738      goto fail;
12739  }
12740  if (mptsas_alloc_free_queue(mpt) == DDI_FAILURE) {
12741      mptsas_log(mpt, CE_WARN, "mptsas_alloc_free_queue failed!");
12742      goto fail;
12743  }
12744  if (mptsas_alloc_post_queue(mpt) == DDI_FAILURE) {
12745      mptsas_log(mpt, CE_WARN, "mptsas_alloc_post_queue failed!");
12746      goto fail;
12747  }
12748  if (mptsas_alloc_reply_frames(mpt) == DDI_FAILURE) {
12749      mptsas_log(mpt, CE_WARN, "mptsas_alloc_reply_frames failed!");
12750      goto fail;
12751  }
12752  mur:
12753  /*
12754  * Re-Initialize ioc to operational state
12755  */
12756  if (mptsas_ioc_init(mpt) == DDI_FAILURE) {
12757      mptsas_log(mpt, CE_WARN, "mptsas_ioc_init failed");
12758      goto fail;
12759  }
12761  mptsas_alloc_reply_args(mpt);
12763  /*

```

```

12759     * Initialize reply post index. Reply free index is initialized after
12760     * the next loop.
12761     */
12762     mpt->m_post_index = 0;

12764     /*
12765     * Initialize the Reply Free Queue with the physical addresses of our
12766     * reply frames.
12767     */
12768     cookie.dmac_address = mpt->m_reply_frame_dma_addr;
12769     for (i = 0; i < mpt->m_max_replies; i++) {
12770         ddi_put32(mpt->m_acc_free_queue_hdl,
12771             &((uint32_t *) (void *) mpt->m_free_queue)[i],
12772             cookie.dmac_address);
12773         cookie.dmac_address += mpt->m_reply_frame_size;
12774     }
12775     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
12776         DDI_DMA_SYNC_FORDEV);

12778     /*
12779     * Initialize the reply free index to one past the last frame on the
12780     * queue. This will signify that the queue is empty to start with.
12781     */
12782     mpt->m_free_index = i;
12783     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex, i);

12785     /*
12786     * Initialize the reply post queue to 0xFFFFFFFF,0xFFFFFFFF's.
12787     */
12788     for (i = 0; i < mpt->m_post_queue_depth; i++) {
12789         ddi_put64(mpt->m_acc_post_queue_hdl,
12790             &((uint64_t *) (void *) mpt->m_post_queue)[i],
12791             0xFFFFFFFFFFFFFFFF);
12792     }
12793     (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
12794         DDI_DMA_SYNC_FORDEV);

12796     /*
12797     * Enable ports
12798     */
12799     if (mptsas_ioc_enable_port(mpt) == DDI_FAILURE) {
12800         mptsas_log(mpt, CE_WARN, "mptsas_ioc_enable_port failed");
12801         goto fail;
12802     }

12804     /*
12805     * enable events
12806     */
12807     if (mptsas_ioc_enable_event_notification(mpt)) {
12808         goto fail;
12809     }

12811     /*
12812     * We need checks in attach and these.
12813     * chip_init is called in mult. places
12814     */

12816     if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
12817         DDI_SUCCESS) ||
12818         (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
12819         DDI_SUCCESS) ||
12820         (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
12821         DDI_SUCCESS) ||
12822         (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
12823         DDI_SUCCESS) ||
12824         (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=

```

```

12825         DDI_SUCCESS)) {
12826             ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
12827             goto fail;
12828         }

12830     /* Check all acc handles */
12831     if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
12832         (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
12833         DDI_SUCCESS) ||
12834         (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
12835         DDI_SUCCESS) ||
12836         (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
12837         DDI_SUCCESS) ||
12838         (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
12839         DDI_SUCCESS) ||
12840         (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
12841         DDI_SUCCESS) ||
12842         (mptsas_check_acc_handle(mpt->m_config_handle) !=
12843         DDI_SUCCESS)) {
12844         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
12845         goto fail;
12846     }

12848     return (DDI_SUCCESS);

12850 fail:
12851     return (DDI_FAILURE);
12852 }
    unchanged portion omitted

14301 void
14302 mptsas_update_driver_data(struct mptsas *mpt)
14303 {
14304     mptsas_target_t *tp;
14305     mptsas_smp_t *sp;

14307     ASSERT(MUTEX_HELD(&mpt->m_mutex));

14309     /*
14310     * TODO after hard reset, update the driver data structures
14311     * 1. update port/phymask mapping table mpt->m_phy_info
14312     * 2. invalid all the entries in hash table
14313     *    m_devhdl = 0xffff and m_deviceinfo = 0
14314     * 3. call sas_device_page/expander_page to update hash table
14315     */
14316     mptsas_update_phymask(mpt);

14318 #endif /* ! codereview */
14319     /*
14320     * Remove all the devhdls for existing entries but leave their
14321     * addresses alone. In update_hashtab() below, we'll find all
14322     * targets that are still present and reassociate them with
14323     * their potentially new devhdls. Leaving the targets around in
14324     * this fashion allows them to be used on the tx waitq even
14325     * while IOC reset is occurring.
14326     * Invalid the existing entries
14327     * XXX - It seems like we should just delete everything here. We are
14328     * holding the lock and are about to refresh all the targets in both
14329     * hashes anyway. Given the path we're in, what outstanding async
14330     * event could possibly be trying to reference one of these things
14331     * without taking the lock, and how would that be useful anyway?
14332     */
14327     for (tp = rehash_first(mpt->m_targets); tp != NULL;
14328         tp = rehash_next(mpt->m_targets, tp)) {
14329         tp->m_devhdl = MPTSAS_INVALID_DEVHDL;

```

```
14330         tp->m_deviceinfo = 0;
14331         tp->m_dr_flag = MPTSAS_DR_INACTIVE;
14332     }
14333     for (sp = rehash_first(mpt->m_smp_targets); sp != NULL;
14334         sp = rehash_next(mpt->m_smp_targets, sp)) {
14335         sp->m_devhdl = MPTSAS_INVALID_DEVHDL;
14336         sp->m_deviceinfo = 0;
14337     }
14338     mpt->m_done_traverse_dev = 0;
14339     mpt->m_done_traverse_smp = 0;
14340     mpt->m_dev_handle = mpt->m_smp_devhdl = MPTSAS_INVALID_DEVHDL;
14341     mptsas_update_hashtab(mpt);
14342 }
```

unchanged portion omitted